Часть 1

Язык Javascript

Илья Кантор

Сборка от 12 января 2021 г.

Последняя версия учебника находится на сайте https://learn.javascript.ru.

Мы постоянно работаем над улучшением учебника. При обнаружении ошибок пишите о них на нашем баг-трекере.

- Введение
  - Введение в JavaScript
  - Справочники и спецификации
  - Редакторы кода
  - Консоль разработчика
- Основы JavaScript
  - Привет, мир!
  - Структура кода
  - Строгий режим — "use strict"
  - Переменные
  - Типы данных
  - Взаимодействие: alert, prompt, confirm
  - Преобразование типов
  - Базовые операторы, математика
  - Операторы сравнения
  - Условное ветвление: if, '?'
  - Логические операторы
  - Оператор объединения с null '??'
  - Циклы while и for
  - Конструкция "switch"
  - Функции
  - Function Expression
  - Функции-стрелки, основы
  - Особенности JavaScript
- Качество кода
  - Отладка в браузере Chrome

- Советы по стилю кода
- Комментарии
- Ниндзя-код
- Автоматическое тестирование с использованием фреймворка Mocha
- Полифилы

Объекты: основы

- Объекты
- Копирование объектов и ссылки
- Сборка мусора
- Методы объекта, "this"
- Конструкторы, создание объектов через "new"
- Опциональная цепочка '?.'
- Тип данных Symbol
- Преобразование объектов в примитивы

Типы данных

- Методы у примитивов
- Числа
- Строки
- Массивы
- Методы массивов
- Перебираемые объекты
- Map и Set
- WeakMap и WeakSet
- Object.keys, values, entries
- Деструктурирующее присваивание
- Дата и время
- Формат JSON, метод toJSON

Продвинутая работа с функциями

- Рекурсия и стек
- Остаточные параметры и оператор расширения
- Замыкание

- Устаревшее ключевое слово "var"
- Глобальный объект
- Объект функции, NFE
- Синтаксис "new Function"
- Планирование: setTimeout и setInterval
- Декораторы и переадресация вызова, call/apply
- Привязка контекста к функции
- Повторяем стрелочные функции

Свойства объекта, их конфигурация

- Флаги и дескрипторы свойств
- Свойства - геттеры и сеттеры

Прототипы, наследование

- Прототипное наследование
- F.prototype
- Встроенные прототипы
- Методы прототипов, объекты без свойства __proto__

Классы

- Класс: базовый синтаксис
- Наследование классов
- Статические свойства и методы
- Приватные и защищённые методы и свойства
- Расширение встроенных классов
- Проверка класса: "instanceof"
- Примеси

Обработка ошибок

- Обработка ошибок, "try..catch"
- Пользовательские ошибки, расширение Error

Промисы, async/await

- Введение: колбэки
- Промисы
- Цепочка промисов

- Промисы: обработка ошибок
- Promise API
- Промисификация
- Микрозадачи
- Async/await

Генераторы, продвинутая итерация
- Генераторы
- Асинхронные итераторы и генераторы

Модули
- Модули, введение
- Экспорт и импорт
- Динамические импорты

Разное
- Proxy и Reflect
- Eval: выполнение строки кода
- Каррирование
- Побитовые операторы
- BigInt
- Intl: интернационализация в JavaScript

Здесь вы можете изучить JavaScript, начиная с нуля и заканчивая продвинутыми концепциями вроде ООП.

Мы сосредоточимся на самом языке, изредка добавляя заметки о средах его исполнения.

Введение

Про язык JavaScript и окружение для разработки на нём.

Введение в JavaScript

Давайте посмотрим, что такого особенного в JavaScript, чего можно достичь с его помощью и какие другие технологии хорошо с ним работают.

Что такое JavaScript?

Изначально *JavaScript* был создан, чтобы *«сделать веб-страницы живыми»*.

Программы на этом языке называются *скриптами*. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Это отличает JavaScript от другого языка – Java.

Почему JavaScript?

Когда JavaScript создавался, у него было другое имя – «LiveScript». Однако, язык Java был очень популярен в то время, и было решено, что позиционирование JavaScript как «младшего брата» Java будет полезно.

Со временем JavaScript стал полностью независимым языком со своей собственной спецификацией, называющейся ECMAScript, и сейчас не имеет никакого отношения к Java.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript.

У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».

Разные движки имеют разные «кодовые имена». Например:

- V8 – в Chrome и Opera.
- SpiderMonkey – в Firefox.
- …Ещё есть «Trident» и «Chakra» для разных версий IE, «ChakraCore» для Microsoft Edge, «Nitro» и «SquirrelFish» для Safari и т.д.

Эти названия полезно знать, так как они часто используются в статьях для разработчиков. Мы тоже будем их использовать. Например, если «функциональность X поддерживается V8», тогда «X», скорее всего, работает в Chrome и Opera.

Как работают движки?

Движки сложны. Но основы понять легко.

1. Движок (встроенный, если это браузер) читает («парсит») текст скрипта.
2. Затем он преобразует («компилирует») скрипт в машинный язык.

3. После этого машинный код запускается и работает достаточно быстро.

Движок применяет оптимизации на каждом этапе. Он даже просматривает скомпилированный скрипт во время его работы, анализируя проходящие через него данные, и применяет оптимизации к машинному коду, полагаясь на полученные знания. В результате скрипты работают очень быстро.

Что может JavaScript в браузере?

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, Node.JS поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

Например, в браузере JavaScript может:

- Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
- Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
- Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (технологии AJAX и COMET).
- Получать и устанавливать куки, задавать вопросы

посетителю, показывать сообщения.

Запоминать данные на стороне клиента («local storage»).

Чего НЕ может JavaScript в браузере?

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя.

Примеры таких ограничений включают в себя:

JavaScript на веб-странице не может читать/записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям ОС.

Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определённые действия, такие как «перетаскивание» файла в окно браузера или его выбор с помощью тега `<input>`.

Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим и отправлять информацию в ФСБ.

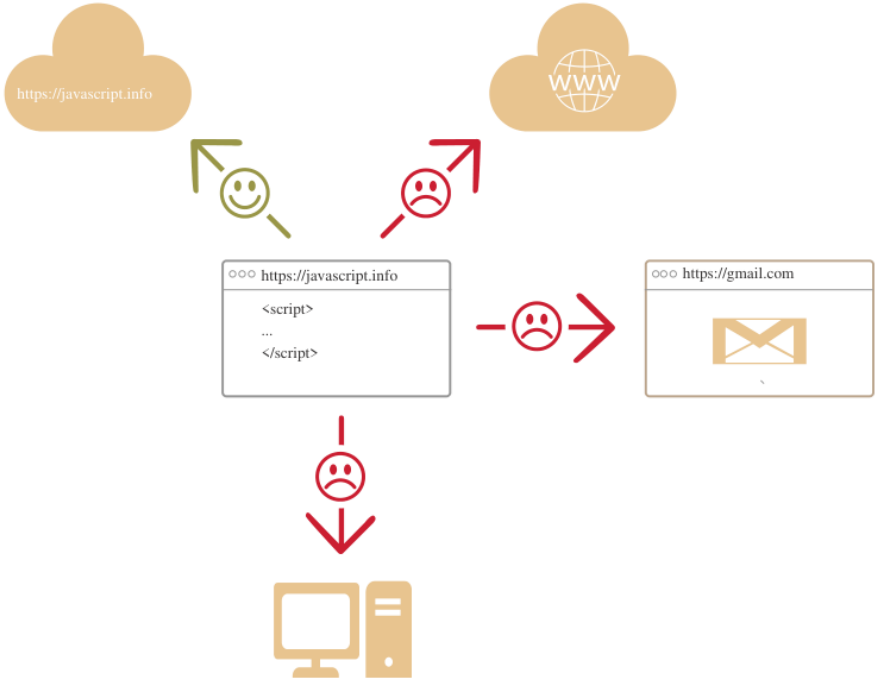
Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или

порта).

Это называется «Политика одинакового источника» (Same Origin Policy). Чтобы обойти это ограничение, обе страницы должны согласиться с этим и содержать JavaScript-код, который специальным образом обменивается данными.

Это ограничение необходимо, опять же, для безопасности пользователя. Страница `https://anysite.com`, которую открыл пользователь, не должна иметь доступ к другой вкладке браузера с URL `https://gmail.com` и воровать информацию оттуда.

JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности.



Подобные ограничения не действуют, если JavaScript используется вне браузера, например — на сервере. Современные браузеры предоставляют плагины/расширения, с помощью которых можно запрашивать дополнительные разрешения.

Что делает JavaScript особенным?

Как минимум, *три* сильные стороны JavaScript:

> Полная интеграция с HTML/CSS.
>
> Простые вещи делаются просто.
>
> Поддерживается всеми основными браузерами и включён по умолчанию.

JavaScript – это единственная браузерная технология, сочетающая в себе все эти три вещи.

Вот что делает JavaScript особенным. Вот почему это самый распространённый инструмент для создания интерфейсов в браузере.

Хотя, конечно, JavaScript позволяет делать приложения не только в браузерах, но и на сервере, на мобильных устройствах и т.п.

Языки «над» JavaScript

Синтаксис JavaScript подходит не под все нужды. Разные люди хотят иметь разные возможности.

Это естественно, потому что проекты разные и требования к ним тоже разные.

Так, в последнее время появилось много новых языков, которые *транспилируются* (конвертируются) в JavaScript, прежде чем запустятся в браузере.

Современные инструменты делают транспиляцию очень быстрой и прозрачной, фактически позволяя разработчикам писать код на другом языке, автоматически преобразуя его в JavaScript «под капотом».

Примеры таких языков:

- CoffeeScript добавляет «синтаксический сахар» для JavaScript. Он вводит более короткий синтаксис, который позволяет писать чистый и лаконичный код. Обычно такое нравится Ruby-программистам.
- TypeScript концентрируется на добавлении «строгой типизации» для упрощения разработки и поддержки больших и сложных систем. Разработан Microsoft.
- Flow тоже добавляет типизацию, но иначе. Разработан

Facebook.

Dart стоит особняком, потому что имеет собственный движок, работающий вне браузера (например, в мобильных приложениях). Первоначально был предложен Google, как замена JavaScript, но на данный момент необходима его транспиляция для запуска так же, как для вышеперечисленных языков.

Brython транспилирует Python в JavaScript, что позволяет писать приложения на чистом Python без JavaScript.

Есть и другие. Но даже если мы используем один из этих языков, мы должны знать JavaScript, чтобы действительно понимать, что мы делаем.

Итого

JavaScript изначально создавался только для браузера, но сейчас используется на многих других платформах.

Сегодня JavaScript занимает уникальную позицию в качестве самого распространённого языка для браузера, обладающего полной интеграцией с HTML/CSS.

Многие языки могут быть «транспилированы» в JavaScript для предоставления дополнительных функций. Рекомендуется хотя бы кратко рассмотреть их после освоения JavaScript.

Справочники и спецификации

Эта книга является *учебником* и нацелена на то, чтобы помочь вам постепенно освоить язык. Но когда вы хорошо изучите основы, вам понадобятся дополнительные источники информации.

Спецификация

Спецификация ECMA-262 содержит самую глубокую, детальную и формализованную информацию о JavaScript. Она определяет сам язык.

Вначале спецификация может показаться тяжеловатой для понимания из-за слишком формального стиля изложения. Если вы ищете источник самой достоверной информации, то это правильное место, но она не для ежедневного использования.

Новая версия спецификации появляется каждый год. А пока она не вышла официально, все желающие могут ознакомиться с текущим черновиком на https://tc39.es/ecma262/.

Чтобы почитать о самых последних возможностях, включая те, которые «почти в стандарте» (так называемые «stage 3 proposals»), посетите https://github.com/tc39/proposals.

Если вы разрабатываете под браузеры, то существуют и другие спецификации, о которых рассказывается во второй части этого учебника.

Справочники

MDN (Mozilla) JavaScript Reference – это справочник с примерами и другой информацией. Хороший источник для получения подробных сведений о функциях языка, методах встроенных объектов и так далее.

Располагается по адресу https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference.

Хотя зачастую вместо их сайта удобнее использовать какой-нибудь интернет-поисковик, вводя там запрос

«MDN [что вы хотите найти]», например https://google.com/search?q=MDN+parseInt для поиска информации о функции `parseInt`.

MSDN – справочник от Microsoft, содержащий много информации, в том числе по JavaScript (который там часто обозначается как JScript). Если вам нужно найти что-то специфическое по браузеру Internet Explorer, лучше искать там: http://msdn.microsoft.com/.

Так же, как и в предыдущем случае, можно использовать интернет-поиск, набирая фразы типа «RegExp MSDN» или «RegExp MSDN jscript».

Таблицы совместимости

JavaScript – это развивающийся язык, в который постоянно добавляется что-то новое.

Посмотреть, какие возможности поддерживаются в разных браузерах и других движках, можно в следующих источниках:

http://caniuse.com – таблицы с информацией о поддержке по каждой возможности языка. Например, чтобы узнать, какие движки поддерживают современные криптографические функции, посетите: http://caniuse.com/#feat=cryptography.

https://kangax.github.io/compat-table – таблица с возможностями языка и движками, которые их поддерживают и не поддерживают.

Все эти ресурсы полезны в ежедневной работе программиста, так как они содержат ценную информацию о возможностях использования языка, их поддержке и так далее.

Пожалуйста, запомните эти ссылки (или ссылку на эту

страницу) на случай, когда вам понадобится подробная информация о какой-нибудь конкретной возможности JavaScript.

Редакторы кода

Большую часть своего рабочего времени программисты проводят в редакторах кода.

Есть два основных типа редакторов: IDE и «лёгкие» редакторы. Многие используют по одному инструменту каждого типа.

IDE

Термином IDE (Integrated Development Environment, «интегрированная среда разработки») называют мощные редакторы с множеством функций, которые работают в рамках целого проекта. Как видно из названия, это не просто редактор, а нечто большее.

IDE загружает проект (который может состоять из множества файлов), позволяет переключаться между файлами, предлагает автодополнение по коду всего проекта (а не только открытого файла), также она интегрирована с системой контроля версий (например, такой как git), средой для тестирования и другими инструментами на уровне всего проекта.

Если вы ещё не выбрали себе IDE, присмотритесь к этим:

- Visual Studio Code (бесплатно).
- WebStorm (платно).

Обе IDE – кроссплатформенные.

Для Windows есть ещё Visual Studio (не путать с Visual Studio Code). Visual Studio – это платная мощная среда разработки, которая работает только на Windows. Она хорошо подходит для .NET платформы. У неё есть бесплатная версия, которая называется Visual Studio Community.

Многие IDE платные, но у них есть пробный период. Их цена обычно незначительна по сравнению с зарплатой квалифицированного разработчика, так что пробуйте и выбирайте ту, что вам подходит лучше других.

«Лёгкие» редакторы

«Лёгкие» редакторы менее мощные, чем IDE, но они отличаются скоростью, удобным интерфейсом и простотой.

В основном их используют для того, чтобы быстро открыть и отредактировать нужный файл.

Главное отличие между «лёгким» редактором и IDE состоит в том, что IDE работает на уровне целого проекта, поэтому она загружает больше данных при запуске, анализирует структуру проекта, если это необходимо, и так далее. Если вы работаете только с одним файлом, то гораздо быстрее открыть его в «лёгком» редакторе.

На практике «лёгкие» редакторы могут иметь множество плагинов, включая автодополнение и анализаторы синтаксиса на уровне директории, поэтому границы между IDE и «лёгкими» редакторами размыты.

Следующие варианты заслуживают вашего внимания:

- Atom (кроссплатформенный, бесплатный).
- Sublime Text (кроссплатформенный, условно-бесплатный).

Notepad++ (Windows, бесплатный).

Vim и Emacs тоже хороши, если знать, как ими пользоваться.

Не будем ссориться

Редакторы, перечисленные выше, известны автору давно и заслужили много хороших отзывов от коллег.

Конечно же, есть много других отличных редакторов. Выбирайте тот, который вам больше нравится.

Выбор редактора, как и любого другого инструмента, индивидуален и зависит от ваших проектов, привычек и личных предпочтений.

Консоль разработчика

Код уязвим для ошибок. И вы, скорее всего, будете делать ошибки в коде… Впрочем, давайте будем откровенны: вы *точно* будете совершать ошибки в коде. В конце концов, вы человек, а не робот.

Но по умолчанию в браузере ошибки не видны. То есть, если что-то пойдёт не так, мы не увидим, что именно сломалось, и не сможем это починить.

Для решения задач такого рода в браузер встроены так называемые «Инструменты разработки» (Developer tools или сокращённо — devtools).

Chrome и Firefox снискали любовь подавляющего большинства программистов во многом благодаря своим отменным инструментам разработчика. Остальные браузеры, хотя и оснащены подобными инструментами, но все же зачастую находятся в роли догоняющих и по качеству, и по количеству свойств и особенностей. В

общем, почти у всех программистов есть свой «любимый» браузер. Другие используются только для отлова и исправления специфичных «браузерозависимых» ошибок.

Для начала знакомства с этими мощными инструментами давайте выясним, как их открывать, смотреть ошибки и запускать команды JavaScript.

Google Chrome

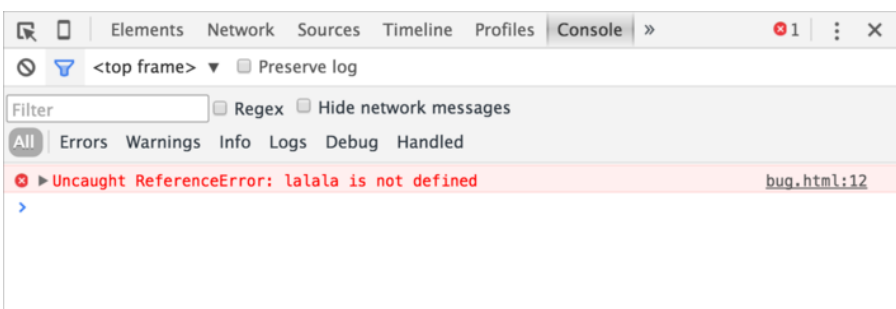
Откройте страницу bug.html.

В её JavaScript-коде закралась ошибка. Она не видна обычному посетителю, поэтому давайте найдём её при помощи инструментов разработки.

Нажмите `F12` или, если вы используете Mac, `Cmd+Opt+J`.

По умолчанию в инструментах разработчика откроется вкладка Console (консоль).

Она выглядит приблизительно следующим образом:



Точный внешний вид инструментов разработки зависит от используемой версии Chrome. Время от времени некоторые детали изменяются, но в целом внешний вид остаётся примерно похожим на предыдущие версии.

> В консоли мы можем увидеть сообщение об ошибке, отрисованное красным цветом. В нашем случае скрипт

содержит неизвестную команду «lalala».

Справа присутствует ссылка на исходный код `bug.html:12` с номером строки кода, в которой эта ошибка и произошла.

Под сообщением об ошибке находится синий символ `>`. Он обозначает командную строку, в ней мы можем редактировать и запускать JavaScript-команды. Для их запуска нажмите `Enter`.

Многострочный ввод

Обычно при нажатии `Enter` введённая строка кода сразу выполняется.

Чтобы перенести строку, нажмите `Shift+Enter`. Так можно вводить более длинный JS-код.

Теперь мы явно видим ошибки, для начала этого вполне достаточно. Мы ещё вернёмся к инструментам разработчика позже и более подробно рассмотрим отладку кода в главе Отладка в браузере Chrome.

Firefox, Edge и другие

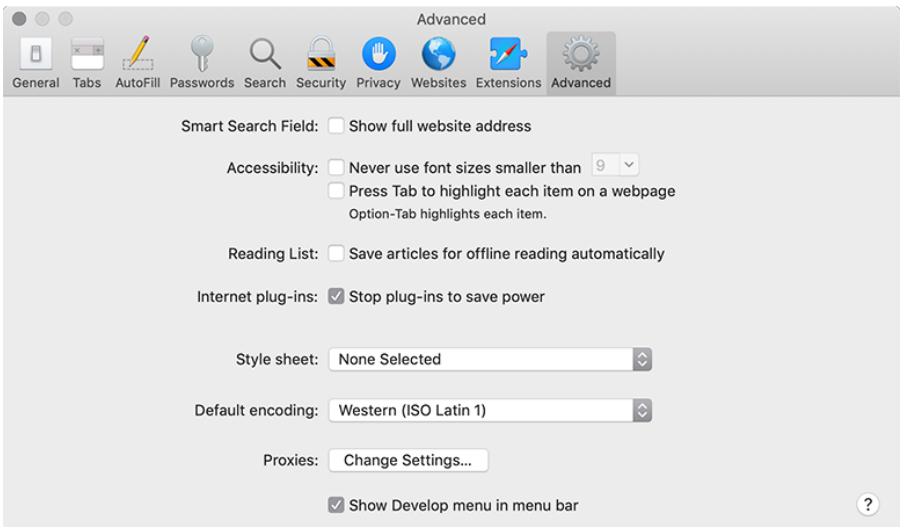
Инструменты разработчика в большинстве браузеров открываются при нажатии на `F12`.

Их внешний вид и принципы работы мало чем отличаются. Разобравшись с инструментами в одном браузере, вы без труда сможете работать с ними и в другом.

Safari

Safari (браузер для Mac, не поддерживается в системах Windows/Linux) всё же имеет небольшое отличие. Для начала работы нам нужно включить «Меню разработки» («Developer menu»).

Откройте Настройки (Preferences) и перейдите к панели «Продвинутые» (Advanced). В самом низу вы найдёте чекбокс:



Теперь консоль можно активировать нажатием клавиш `Cmd+Opt+C`. Также обратите внимание на новый элемент меню «Разработка» («Develop»). В нем содержится большое количество команд и настроек.

Итого

> Инструменты разработчика позволяют нам смотреть ошибки, выполнять команды, проверять значение переменных и ещё много всего полезного.
>
> В большинстве браузеров, работающих под Windows, инструменты разработчика можно открыть, нажав `F12`. В Chrome для Mac используйте комбинацию `Cmd+Opt+J`, Safari: `Cmd+Opt+C` (необходимо предварительное

включение «Меню разработчика»).

Теперь наше окружение полностью настроено. В следующем разделе мы перейдём непосредственно к JavaScript.

Основы JavaScript

Давайте изучим основы создания скриптов.

Привет, мир!

В этой части учебника мы изучаем собственно JavaScript, сам язык.

Но нам нужна рабочая среда для запуска наших скриптов, и, поскольку это онлайн-книга, то браузер будет хорошим выбором. В этой главе мы сократим количество специфичных для браузера команд (например, `alert`) до минимума, чтобы вы не тратили на них время, если планируете сосредоточиться на другой среде (например, Node.js). А на использовании JavaScript в браузере мы сосредоточимся в следующей части учебника.

Итак, сначала давайте посмотрим, как выполнить скрипт на странице. Для серверных сред (например, Node.js), вы можете выполнить скрипт с помощью команды типа `"node my.js"`. Для браузера всё немного иначе.

Тег «script»

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега `<script>`.

Для примера:

```
<!DOCTYPE HTML>
<html>

<body>
```

```
  <p>Перед скриптом...</p>


  <p>...После скрипта.</p>

</body>

</html>
```

```
  <script>
    alert( 'Привет, мир!' );
  </script>
```

Тег `<script>` содержит JavaScript-код, который автоматически выполнится, когда браузер его обработает.

Современная разметка

Тег `<script>` имеет несколько атрибутов, которые редко используются, но всё ещё могут встретиться в старом коде:

Атрибут `type`: `<script type=…>`

Старый стандарт HTML, HTML4, требовал наличия этого атрибута в теге `<script>`. Обычно он имел значение `type="text/javascript"`. На текущий момент этого больше не требуется. Более того, в современном стандарте HTML смысл этого атрибута полностью изменился. Теперь он может использоваться для JavaScript-модулей. Но это тема не для начального уровня, и о ней мы поговорим в другой части учебника.

Атрибут `language`: `<script language=…>`

Этот атрибут должен был задавать язык, на котором написан скрипт. Но так как JavaScript является языком по умолчанию, в этом атрибуте уже нет необходимости.

Обёртывание скрипта в HTML-комментарии.

В очень древних книгах и руководствах вы сможете найти комментарии внутри тега `<script>`, например, такие:

```
<script type="text/javascript"><!--
    ...
//--></script>
```

Этот комментарий скрывал код JavaScript в старых браузерах, которые не знали, как обрабатывать тег `<script>`. Поскольку все браузеры, выпущенные за последние 15 лет, не содержат данной проблемы, такие комментарии уже не нужны. Если они есть, то это признак, что перед нами очень древний код.

Внешние скрипты

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл.

Файл скрипта можно подключить к HTML с помощью атрибута `src`:

```
<script src="/path/to/script.js"></script>
```

Здесь `/path/to/script.js` – это абсолютный путь до скрипта от корня сайта. Также можно указать относительный путь от текущей страницы. Например, `src="script.js"` будет означать, что файл `"script.js"` находится в текущей папке.

Можно указать и полный URL-адрес. Например:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/
lodash.js/3.2.0/lodash.js"></script>
```

Для подключения нескольких скриптов используйте несколько тегов:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
…
```

На заметку:

Как правило, только простейшие скрипты помещаются в HTML. Более сложные выделяются в отдельные файлы.

Польза от отдельных файлов в том, что браузер загрузит скрипт отдельно и сможет хранить его в кеше.

Другие страницы, которые подключают тот же скрипт, смогут брать его из кеша вместо повторной загрузки из сети. И таким образом файл будет загружаться с сервера только один раз.

Это сокращает расход трафика и ускоряет загрузку страниц.

Если атрибут `src` установлен, содержимое тега `script` будет игнорироваться.

В одном теге `<script>` нельзя использовать одновременно атрибут `src` и код внутри.

Нижеприведённый пример не работает:

```
<script ="file.js">
  alert(1); // содержимое игнорируется, так как есть атрибут src
</script>
```

src

Нужно выбрать: либо внешний скрипт `<script src="…">`, либо обычный код внутри тега `<script>`.

Вышеприведённый пример можно разделить на два скрипта:

```
<script src="file.js"></script>
<script>
  alert(1);
</script>
```

Итого

Для добавления кода JavaScript на страницу используется тег `<script>`

Атрибуты `type` и `language` необязательны.

Скрипт во внешнем файле можно вставить с помощью `<script src="path/to/script.js"></script>`.

Нам ещё многое предстоит изучить про браузерные скрипты и их взаимодействие со страницей. Но, как уже было сказано, эта часть учебника посвящена именно языку JavaScript, поэтому здесь мы постараемся не отвлекаться на детали реализации в браузере. Мы воспользуемся браузером для запуска JavaScript, это удобно для онлайн-демонстраций, но это только одна из платформ, на которых работает этот язык.

Задачи

Вызвать alert

важность: 5

Создайте страницу, которая отобразит сообщение «Я JavaScript!».

Выполните это задание в песочнице, либо на вашем жёстком диске, где – неважно, главное – проверьте, что она работает.

Демо в новом окне

К решению

Покажите сообщение с помощью внешнего скрипта

важность: 5

Возьмите решение предыдущей задачи Вызвать alert, и измените его. Извлеките содержимое скрипта во внешний файл `alert.js`, лежащий в той же папке.

Откройте страницу, убедитесь, что оповещение работает.

К решению

Структура кода

Начнём изучение языка с рассмотрения основных

«строительных блоков» кода.

Инструкции

Инструкции – это синтаксические конструкции и команды, которые выполняют действия.

Мы уже видели инструкцию `alert('Привет, мир!')`, которая отображает сообщение «Привет, мир!».

В нашем коде может быть столько инструкций, сколько мы захотим. Инструкции могут отделяться точкой с запятой.

Например, здесь мы разделили сообщение «Привет Мир» на два вызова alert:

```
alert('Привет'); alert('Мир');
```

Обычно каждую инструкцию пишут на новой строке, чтобы код было легче читать:

```
alert('Привет');
alert('Мир');
```

Точка с запятой

В большинстве случаев точку с запятой можно не ставить, если есть переход на новую строку.

Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

В этом случае JavaScript интерпретирует перенос строки как «неявную» точку с запятой. Это называется

автоматическая вставка точки с запятой.

В большинстве случаев новая строка подразумевает точку с запятой. Но «в большинстве случаев» не значит «всегда»!

В некоторых ситуациях новая строка всё же не означает точку с запятой. Например:

```
alert(3 +
1
+ 2);
```

Код выведет `6`, потому что JavaScript не вставляет здесь точку с запятой. Интуитивно очевидно, что, если строка заканчивается знаком `"+"`, значит, это «незавершённое выражение», поэтому точка с запятой не требуется. И в этом случае всё работает, как задумано.

Но есть ситуации, где JavaScript «забывает» вставить точку с запятой там, где она нужна.

Ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Пример ошибки

Если вы хотите увидеть конкретный пример такой ошибки, обратите внимание на этот код:

```
[1, 2].forEach(alert)
```

Пока нет необходимости знать значение скобок `[]` и `forEach`. Мы изучим их позже. Пока что просто запомните результат выполнения этого кода: выводится `1`, а затем `2`.

А теперь добавим `alert` перед кодом и *не* поставим в конце точку с запятой:

```
alert("Сейчас будет ошибка")

[1, 2].forEach(alert)
```

Теперь, если запустить код, выведется только первый `alert`, а затем мы получим ошибку!

Всё исправится, если мы поставим точку с запятой после `alert`:

```
alert("Теперь всё в порядке");

[1, 2].forEach(alert)
```

Теперь мы получим сообщение «Теперь всё в порядке», следом за которым будут `1` и `2`.

В первом примере без точки с запятой возникает ошибка, потому что JavaScript не вставляет точку с запятой перед квадратными скобками `[...]`. И поэтому код в первом примере выполняется, как одна инструкция. Вот как движок видит его:

```
alert("Сейчас будет ошибка")[1, 2].forEach(alert)
```

Но это должны быть две отдельные инструкции, а не одна. Такое слияние в данном случае неправильное, оттого и ошибка. Это может произойти и в некоторых других ситуациях.

Мы рекомендуем ставить точку с запятой между инструкциями, даже если они отделены переносами строк. Это правило широко используется в сообществе разработчиков. Стоит отметить ещё раз – в большинстве случаев *можно* не ставить точку с запятой. Но безопаснее,

особенно для новичка, ставить её.

Комментарии

Со временем программы становятся всё сложнее и сложнее. Возникает необходимость добавлять *комментарии*, которые бы описывали, что делает код и почему.

Комментарии могут находиться в любом месте скрипта. Они не влияют на его выполнение, поскольку движок просто игнорирует их.

Однострочные комментарии начинаются с двойной косой черты `//`.

Часть строки после `//` считается комментарием. Такой комментарий может как занимать строку целиком, так и находиться после инструкции.

Как здесь:

```
// Этот комментарий занимает всю строку
alert('Привет');

alert('Мир'); // Этот комментарий следует за инструкцией
```

Многострочные комментарии начинаются косой чертой со звёздочкой `/*` и заканчиваются звёздочкой с косой чертой `*/`.

Как вот здесь:

```
/* Пример с двумя сообщениями.
Это – многострочный комментарий.
*/
alert('Привет');
alert('Мир');
```

Содержимое комментария игнорируется, поэтому, если мы поместим код внутри `/* … */`, он не будет исполняться.

Это бывает удобно для временного отключения участка кода:

```
/* Закомментировали код
alert('Привет');
*/
alert('Мир');
```

Используйте горячие клавиши!

В большинстве редакторов строку кода можно закомментировать, нажав комбинацию клавиш `Ctrl+/` для однострочного комментария и что-то вроде `Ctrl+Shift+/` – для многострочных комментариев (выделите кусок кода и нажмите комбинацию клавиш). В системе Mac попробуйте `Cmd` вместо `Ctrl` и `Option` вместо `Shift`.

Вложенные комментарии не поддерживаются!

Не может быть `/*...*/` внутри `/*...*/`.

Такой код «умрёт» с ошибкой:

```
/*
  /* вложенный комментарий ?!? */
*/
alert( 'Мир' );
```

Не стесняйтесь использовать комментарии в своём коде.

Комментарии увеличивают размер кода, но это не проблема. Есть множество инструментов, которые минифицируют код перед публикацией на рабочий сервер. Они убирают комментарии, так что они не содержатся в рабочих скриптах. Таким образом, комментарии никоим образом не вредят рабочему коду.

Позже в учебнике будет глава Качество кода, которая объяснит, как лучше писать комментарии.

Строгий режим — "use strict"

На протяжении долгого времени JavaScript развивался без проблем с обратной совместимостью. Новые функции добавлялись в язык, в то время как старая функциональность не менялась.

Преимуществом данного подхода было то, что существующий код продолжал работать. А недостатком – что любая ошибка или несовершенное решение, принятое создателями JavaScript, застревали в языке навсегда.

Так было до 2009 года, когда появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: `"use strict"`.

«use strict»

Директива выглядит как строка: `"use strict"` или `'use strict'`. Когда она находится в начале скрипта, весь сценарий работает в «современном» режиме.

Например:

```
"use strict";

// этот код работает в современном режиме
...
```

Позже мы изучим функции (способ группировки команд). Забегая вперёд, заметим, что вместо всего скрипта `"use strict"` можно поставить в начале большинства видов функций. Это позволяет включить строгий режим только в конкретной функции. Но обычно люди используют его для всего файла.

Убедитесь, что «use strict» находится в начале

Проверьте, что `"use strict"` находится в первой исполняемой строке скрипта, иначе строгий режим может не включиться.

Здесь строгий режим не включён:

```
alert("some code");
// "use strict" ниже игнорируется – он должен быть в первой строке

"use strict";

// строгий режим не активирован
```

Над `"use strict"` могут быть записаны только комментарии.

Нет никакого способа отменить `use strict`

Нет директивы типа `"no use strict"`, которая

возвращала бы движок к старому поведению.

Как только мы входим в строгий режим, отменить это невозможно.

Консоль браузера

В дальнейшем, когда вы будете использовать консоль браузера для тестирования функций, обратите внимание, что `use strict` по умолчанию в ней выключен.

Иногда, когда `use strict` имеет значение, вы можете получить неправильные результаты.

Можно использовать `Shift+Enter` для ввода нескольких строк и написать в верхней строке `use strict`:

```
'use strict'; <Shift+Enter для перехода на новую строку>
//  ...ваш код...
<Enter для запуска>
```

В большинстве браузеров, включая Chrome и Firefox, это работает.

В старых браузерах консоль не учитывает такой `use strict`, там можно «оборачивать» код в функцию, вот так:

```
(function() {
  'use strict';

  // ...ваш код...
})()
```

Всегда ли нужно использовать «use strict»?

Вопрос кажется риторическим, но это не так.

Кто-то посоветует начинать каждый скрипт с `"use strict"`... Но есть способ покруче.

Современный JavaScript поддерживает «классы» и «модули» — продвинутые структуры языка (и мы, конечно, до них доберёмся), которые автоматически включают строгий режим. Поэтому в них нет нужды добавлять директиву `"use strict"`.

Подытожим: пока очень желательно добавлять `"use strict";` в начале ваших скриптов. Позже, когда весь ваш код будет состоять из классов и модулей, директиву можно будет опускать.

Пока мы узнали о `use strict` только в общих чертах.

В следующих главах, по мере расширения знаний о возможностях языка, мы яснее увидим отличия между строгим и стандартным режимом. К счастью, их не так много, и все они делают жизнь разработчика лучше.

Все примеры в этом учебнике подразумевают исполнение в строгом режиме, за исключением случаев (очень редких), когда оговорено иное.

Переменные

JavaScript-приложению обычно нужно работать с информацией. Например:

1. Интернет-магазин – информация может включать продаваемые товары и корзину покупок.
2. Чат – информация может включать пользователей, сообщения и многое другое.

Переменные используются для хранения этой информации.

Переменная

Переменная – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово `let`.

Приведённая ниже инструкция создаёт (другими словами: *объявляет* или *определяет*) переменную с именем «message»:

```
let message;
```

Теперь можно поместить в неё данные, используя оператор присваивания `=`:

```
let message;

message = 'Hello'; // сохранить строку
```

Строка сохраняется в области памяти, связанной с переменной. Мы можем получить к ней доступ, используя имя переменной:

```
let message;
message = 'Hello!';

alert(message); // показывает содержимое
переменной
```

Для краткости можно совместить объявление переменной и запись данных в одну строку:

```
let message = 'Hello!'; // определяем переменную и присваиваем
ей значение
```

```
alert(message); // Hello!
```

Мы также можем объявить несколько переменных в одной строке:

```
let user = 'John', age = 25, message = 'Hello';
```

Такой способ может показаться короче, но мы не рекомендуем его. Для лучшей читаемости объявляйте каждую переменную на новой строке.

Многострочный вариант немного длиннее, но легче для чтения:

```
let user = 'John';
let age = 25;
let message = 'Hello';
```

Некоторые люди также определяют несколько переменных в таком вот многострочном стиле:

```
let user = 'John',
  age = 25,
  message = 'Hello';
```

...Или даже с запятой в начале строки:

```
let user = 'John'
  , age = 25
  , message = 'Hello';
```

В принципе, все эти варианты работают одинаково. Так что это вопрос личного вкуса и эстетики.

`var` вместо `let`

В старых скриптах вы также можете найти другое ключевое слово: `var` вместо `let`:

```
message = 'Hello';
var
```

Ключевое слово `var` – *почти* то же самое, что и `let`. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

Есть тонкие различия между `let` и `var`, но они пока не имеют для нас значения. Мы подробно рассмотрим их в главе Устаревшее ключевое слово "var".

Аналогия из жизни

Мы легко поймём концепцию «переменной», если представим её в виде «коробки» для данных с уникальным названием на ней.

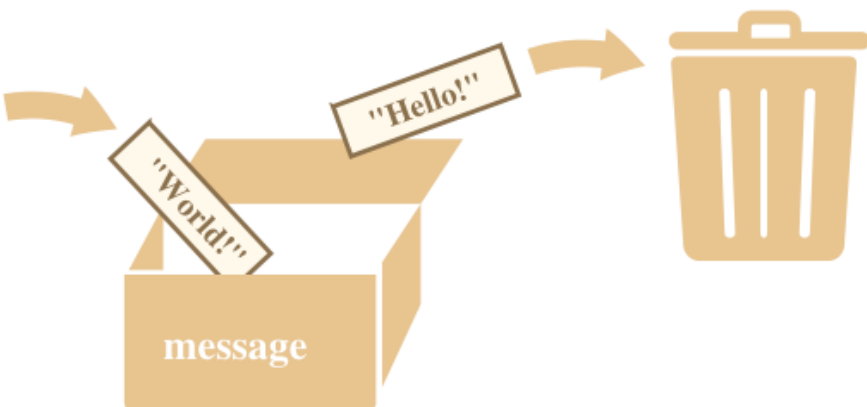
Например, переменную `message` можно представить как коробку с названием `"message"` и значением `"Hello!"` внутри:

Мы можем положить любое значение в коробку.

Мы также можем изменить его столько раз, сколько захотим:

```
let message;

message = 'Hello!';

message = 'World!'; // значение изменено

alert(message);
```

При изменении значения старые данные удаляются из переменной:



Мы также можем объявить две переменные и скопировать данные из одной в другую.

```
let hello = 'Hello world!';

let message;


// теперь две переменные содержат одинаковые данные
alert(hello); // Hello world!
alert(message); // Hello world!
// копируем значение 'Hello world' из переменной hello в переменную message
message = hello;
```

Повторное объявление вызывает ошибку

Переменная может быть объявлена только один раз.

Повторное объявление той же переменной является ошибкой:

```
let message = "Это";

// повторение ключевого слова 'let' приводит к ошибке
let message = "Другое"; // SyntaxError: 'message' has already been declared
```

Поэтому следует объявлять переменную только один раз и затем использовать её уже без `let`.

Функциональные языки программирования

Примечательно, что существуют функциональные языки программирования, такие как Scala или Erlang, которые запрещают изменять значение переменной.

В таких языках однажды сохранённое «в коробку» значение остаётся там навсегда. Если нам нужно сохранить что-то другое, язык заставляет нас создать новую коробку (объявить новую переменную). Мы не можем использовать старую переменную.

Хотя на первый взгляд это может показаться немного странным, эти языки вполне подходят

для серьёзной разработки. Более того, есть такая область, как параллельные вычисления, где это ограничение даёт определённые преимущества. Изучение такого языка (даже если вы не планируете использовать его в ближайшее время) рекомендуется для расширения кругозора.

Имена переменных

В JavaScript есть два ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы `$` и `_`.
2. Первый символ не должен быть цифрой.

Примеры допустимых имён:

```
let userName;
let test123;
```

Если имя содержит несколько слов, обычно используется верблюжья нотация, то есть, слова следуют одно за другим, где каждое следующее слово начинается с заглавной буквы: `myVeryLongName`.

Самое интересное – знак доллара `'$'` и подчёркивание `'_'` также можно использовать в названиях. Это обычные символы, как и буквы, без какого-либо особого значения.

Эти имена являются допустимыми:

```
let $ = 1; // объявили переменную с именем "$"
let _ = 2; // а теперь переменную с именем "_"

alert($ + _); // 3
```

Примеры неправильных имён переменных:

```
let 1a; // не может начинаться с цифры

let my-name; // дефис '-' не разрешён в имени
```

Регистр имеет значение

Переменные с именами `apple` и `AppLE` – это две разные переменные.

Нелатинские буквы разрешены, но не рекомендуются

Можно использовать любой язык, включая кириллицу или даже иероглифы, например:

```
let имя = '...';
let 我 = '...';
```

Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных. Даже если мы пишем небольшой скрипт, у него может быть долгая жизнь впереди. Людям из других стран, возможно, придётся прочесть его не один раз.

Зарезервированные имена

Существует список зарезервированных слов, которые нельзя использовать в качестве имён

переменных, потому что они используются самим языком.

Например: `let`, `class`, `return` и `function` зарезервированы.

Приведённый ниже код даёт синтаксическую ошибку:

```
let let = 5; // нельзя назвать переменную "let", ошибка!
let return = 5; // также нельзя назвать переменную "return", ошибка!
```

Создание переменной без использования `use strict`

Обычно нам нужно определить переменную перед её использованием. Но в старые времена было технически возможно создать переменную простым присвоением значения без использования `let`. Это все ещё работает, если мы не включаем `use strict` в наших файлах, чтобы обеспечить совместимость со старыми скриптами.

```
// заметка: "use strict" в этом примере не используется

num = 5; // если переменная "num" раньше не существовала, она создаётся

alert(num); // 5
```

Это плохая практика, которая приводит к ошибке в строгом режиме:

```
"use strict";
```

```
num = 5; // ошибка: num is not defined
```

Константы

Чтобы объявить константную, то есть, неизменяемую переменную, используйте `const` вместо `let`:

```
const myBirthday = '18.04.1982';
```

Переменные, объявленные с помощью `const`, называются «константами». Их нельзя изменить. Попытка сделать это приведёт к ошибке:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // ошибка, константу нельзя перезаписать!
```

Если программист уверен, что переменная никогда не будет меняться, он может гарантировать это и наглядно донести до каждого, объявив её через `const`.

Константы в верхнем регистре

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта.

Названия таких констант пишутся с использованием заглавных букв и подчёркивания.

Например, сделаем константы для различных цветов в «шестнадцатеричном формате»:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
```

```
const COLOR_ORANGE = "#FF7F00";

// ...когда нам нужно выбрать цвет
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Преимущества:

- `COLOR_ORANGE` гораздо легче запомнить, чем `"#FF7F00"`.
- Гораздо легче допустить ошибку при вводе `"#FF7F00"`, чем при вводе `COLOR_ORANGE`.
- При чтении кода `COLOR_ORANGE` намного понятнее, чем `#FF7F00`.

Когда мы должны использовать для констант заглавные буквы, а когда называть их нормально? Давайте разберёмся и с этим.

Название «константа» просто означает, что значение переменной никогда не меняется. Но есть константы, которые известны до выполнения (например, шестнадцатеричное значение для красного цвета), а есть константы, которые *вычисляются* во время выполнения сценария, но не изменяются после их первоначального назначения.

Например:

```
const pageLoadTime = /* время, потраченное на загрузку веб-страницы */;
```

Значение `pageLoadTime` неизвестно до загрузки страницы, поэтому её имя записано обычными, а не прописными буквами. Но это всё ещё константа, потому что она не изменяется после назначения.

Другими словами, константы с именами, записанными заглавными буквами, используются только как псевдонимы

для «жёстко закодированных» значений.

Придумывайте правильные имена

В разговоре о переменных необходимо упомянуть, что есть ещё одна чрезвычайно важная вещь.

Название переменной должно иметь ясный и понятный смысл, говорить о том, какие данные в ней хранятся.

Именование переменных – это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой – опытным разработчиком.

В реальном проекте большая часть времени тратится на изменение и расширение существующей кодовой базы, а не на написание чего-то совершенно нового с нуля. Когда мы возвращаемся к коду после какого-то промежутка времени, гораздо легче найти информацию, которая хорошо размечена. Или, другими словами, когда переменные имеют хорошие имена.

Пожалуйста, потратьте время на обдумывание правильного имени переменной перед её объявлением. Делайте так, и будете вознаграждены.

Несколько хороших правил:

- Используйте легко читаемые имена, такие как `userName` или `shoppingCart`.
- Избегайте использования аббревиатур или коротких имён, таких как `a`, `b`, `c`, за исключением тех случаев, когда вы точно знаете, что так нужно.
- Делайте имена максимально описательными и лаконичными. Примеры плохих имён: `data` и `value`. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно,

какие данные хранит переменная.

Договоритесь с вашей командой об используемых терминах. Если посетитель сайта называется «user», тогда мы должны называть связанные с ним переменные `currentUser` или `newUser`, а не, к примеру, `currentVisitor` или `newManInTown`.

Звучит просто? Действительно, это так, но на практике для создания описательных и кратких имён переменных зачастую требуется подумать. Действуйте.

Повторно использовать или создавать новую переменную?

И последняя заметка. Есть ленивые программисты, которые вместо объявления новых переменных повторно используют существующие.

В результате их переменные похожи на коробки, в которые люди бросают разные предметы, не меняя на них этикетки. Что сейчас находится внутри коробки? Кто знает? Нам необходимо подойти поближе и проверить.

Такие программисты немного экономят на объявлении переменных, но теряют в десять раз больше при отладке.

Дополнительная переменная – это добро, а не зло.

Современные JavaScript-минификаторы и браузеры оптимизируют код достаточно хорошо, поэтому он не создаёт проблем с производительностью. Использование разных переменных для разных значений может даже

помочь движку оптимизировать ваш код.

Итого

Мы можем объявить переменные для хранения данных с помощью ключевых слов `var`, `let` или `const`.

- `let` – это современный способ объявления.
- `var` – это устаревший способ объявления. Обычно мы вообще не используем его, но мы рассмотрим тонкие отличия от `let` в главе Устаревшее ключевое слово "var" на случай, если это всё-таки вам понадобится.
- `const` – похоже на `let`, но значение переменной не может изменяться.

Переменные должны быть названы таким образом, чтобы мы могли легко понять, что у них внутри.

Задачи

Работа с переменными

важность: 2

1. Объявите две переменные: `admin` и `name`.
2. Запишите строку `"Джон"` в переменную `name`.
3. Скопируйте значение из переменной `name` в `admin`.
4. Выведите на экран значение `admin`, используя функцию `alert` (должна показать «Джон»).

К решению

Придумайте правильные имена

важность: 3

1. Создайте переменную для названия нашей планеты. Как бы вы её назвали?
2. Создайте переменную для хранения имени текущего посетителя сайта. Как бы вы назвали такую переменную?

К решению

Какие буквы (заглавные или строчные) использовать для имён констант?

важность: 4

Рассмотрим следующий код:

```
const birthday = '18.04.1982';

const age = someCode(birthday);
```

У нас есть константа `birthday`, а также `age`, которая вычисляется при помощи некоторого кода, используя значение из `birthday` (в данном случае детали не имеют значения, поэтому код не рассматривается).

Можно ли использовать заглавные буквы для имени `birthday`? А для `age`? Или одновременно для обеих переменных?

```
const BIRTHDAY = '18.04.1982'; // использовать заглавные буквы?
```

```
const AGE = someCode(BIRTHDAY); // а здесь?
```

К решению

Типы данных

Значение в JavaScript всегда относится к данным определённого типа. Например, это может быть строка или число.

Есть восемь основных типов данных в JavaScript. В этой главе мы рассмотрим их в общем, а в следующих главах поговорим подробнее о каждом.

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

```
// Не будет ошибкой
let message = "hello";
message = 123456;
```

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

Число

```
let n = 123;
n = 12.345;
```

Числовой тип данных (`number`) представляет как целочисленные значения, так и числа с плавающей точкой.

Существует множество операций для чисел, например, умножение `*`, деление `/`, сложение `+`, вычитание `-` и так далее.

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: `Infinity`, `-Infinity` и `NaN`.

`Infinity` представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа.

Мы можем получить его в результате деления на ноль:

```
alert( 1 / 0 ); // Infinity
```

Или задать его явно:

```
alert( Infinity ); // Infinity
```

`NaN` означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

```
alert( "не число" / 2 ); // NaN, такое деление является ошибкой
```

Значение `NaN` «прилипчиво». Любая операция с `NaN` возвращает `NaN`:

```
alert( "не число" / 2 + 5 ); // NaN
```

Если где-то в математическом выражении есть `NaN`, то результатом вычислений с его участием будет `NaN`.

Математические операции – безопасны

Математические операции в JavaScript «безопасны». Мы можем делать что угодно: делить на ноль, обращаться с нечисловыми строками как с числами и т.д.

Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»). В худшем случае мы получим `NaN` как результат выполнения.

Специальные числовые значения относятся к типу «число». Конечно, это не числа в привычном значении этого слова.

Подробнее о работе с числами мы поговорим в главе Числа.

BigInt

В JavaScript тип «number» не может содержать числа больше, чем $(2^{53}-1)$ (т. е. `9007199254740991`), или меньше, чем $-(2^{53}-1)$ для отрицательных чисел. Это техническое ограничение вызвано их внутренним представлением.

Для большинства случаев этого достаточно. Но иногда нам нужны действительно гигантские числа, например, в криптографии или при использовании метки времени («timestamp») с микросекундами.

Тип `BigInt` был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа `BigInt`, необходимо добавить `n` в конец числового литерала:

```
// символ "n" в конце означает, что это BigInt
const bigInt = 1234567890123456789012345678901234567890n;
```

Так как `BigInt`-числа нужны достаточно редко, мы рассмотрим их в отдельной главе BigInt. Ознакомьтесь с ней, когда вам понадобятся настолько большие числа.

> **Поддержка**
>
> В данный момент `BigInt` поддерживается только в браузерах Firefox, Chrome, Edge и Safari, но не поддерживается в IE.

Строка

Строка (`string`) в JavaScript должна быть заключена в кавычки.

```
let str = "Привет";
let str2 = 'Одинарные кавычки тоже подойдут';
let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;
```

В JavaScript существует три типа кавычек.

1. Двойные кавычки: `"Привет"`.
2. Одинарные кавычки: `'Привет'`.
3. Обратные кавычки: `` `Привет` ``.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в `${…}`. Например:

```
let name = "Иван";

// Вставим переменную
alert( `Привет, !` ); // Привет, Иван!

// Вставим выражение
alert( `результат: ` ); // результат: 3
${
name}
${
1 + 2}
```

Выражение внутри `${…}` вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную `name`, или выражение `1 + 2`, или что-то более сложное.

Обратите внимание, что это можно делать только в обратных кавычках. Другие кавычки не имеют такой функциональности встраивания!

```
alert( "результат: ${1 + 2}" ); // результат: ${1 + 2}
(двойные кавычки ничего не делают)
```

Мы рассмотрим строки более подробно в главе Строки.

Нет отдельного типа данных для одного символа.

В некоторых языках, например C и Java, для хранения одного символа, например `"a"` или `"%"`, существует отдельный тип. В языках C и Java это `char`.

В JavaScript подобного типа нет, есть только тип `string`. Строка может содержать ноль символов (быть пустой), один символ или множество.

Булевый (логический) тип

Булевый тип (`boolean`) может принимать только два значения: `true` (истина) и `false` (ложь).

Такой тип, как правило, используется для хранения значений да/нет: `true` значит «да, правильно», а `false` значит «нет, не правильно».

Например:

```
let nameFieldChecked = true; // да, поле отмечено
let ageFieldChecked = false; // нет, поле не отмечено
```

Булевые значения также могут быть результатом сравнений:

```
let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")
```

Мы рассмотрим булевые значения более подробно в главе Логические операторы.

Значение «null»

Специальное значение `null` не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение `null`:

```
let age = null;
```

В JavaScript `null` не является «ссылкой на несуществующий объект» или «нулевым указателем», как

в некоторых других языках.

Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

В приведённом выше коде указано, что значение переменной `age` неизвестно.

Значение «undefined»

Специальное значение `undefined` также стоит особняком. Оно формирует тип из самого себя так же, как и `null`.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет `undefined`:

```
let age;

alert(age); // выведет "undefined"
```

Технически мы можем присвоить значение `undefined` любой переменной:

```
let age = 123;

// изменяем значение на undefined
age = undefined;

alert(age); // "undefined"
```

…Но так делать не рекомендуется. Обычно `null` используется для присвоения переменной «пустого» или «неизвестного» значения, а `undefined` – для проверок, была ли переменная назначена.

Объекты и символы

Тип `object` (объект) – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка, или число, или что-то ещё). В объектах же хранят коллекции данных или более сложные структуры.

Объекты занимают важное место в языке и требуют особого внимания. Мы разберёмся с ними в главе Объекты после того, как узнаем больше о примитивах.

Тип `symbol` (символ) используется для создания уникальных идентификаторов в объектах. Мы упоминаем здесь о нём для полноты картины, изучим этот тип после объектов.

Оператор typeof

Оператор `typeof` возвращает тип аргумента. Это полезно, когда мы хотим обрабатывать значения различных типов по-разному или просто хотим сделать проверку.

У него есть две синтаксические формы:

1. Синтаксис оператора: `typeof x`.
2. Синтаксис функции: `typeof(x)`.

Другими словами, он работает со скобками или без скобок. Результат одинаковый.

Вызов `typeof x` возвращает строку с именем типа:

```
typeof undefined // "undefined"

typeof 0 // "number"
```

```
typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"
```

```
typeof Math // "object"  (1)
typeof null // "object"  (2)
typeof alert // "function"  (3)
```

Последние три строки нуждаются в пояснении:

1. `Math` — это встроенный объект, который предоставляет математические операции и константы. Мы рассмотрим его подробнее в главе Числа. Здесь он служит лишь примером объекта.
2. Результатом вызова `typeof null` является `"object"`. Это официально признанная ошибка в `typeof`, ведущая начало с времён создания JavaScript и сохранённая для совместимости. Конечно, `null` не является объектом. Это специальное значение с отдельным типом.
3. Вызов `typeof alert` возвращает `"function"`, потому что `alert` является функцией. Мы изучим функции в следующих главах, где заодно увидим, что в JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но `typeof` обрабатывает их особым образом, возвращая `"function"`. Так тоже повелось от создания JavaScript. Формально это неверно, но может быть удобным на практике.

Итого

В JavaScript есть 8 основных типов.

`number` для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном $\pm(2^{53}-1)$.

`bigint` для целых чисел произвольной длины.

`string` для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.

`boolean` для `true`/`false`.

`null` для неизвестных значений – отдельный тип, имеющий одно значение `null`.

`undefined` для неприсвоенных значений – отдельный тип, имеющий одно значение `undefined`.

`object` для более сложных структур данных.

`symbol` для уникальных идентификаторов.

Оператор `typeof` позволяет нам увидеть, какой тип данных сохранён в переменной.

Имеет две формы: `typeof x` или `typeof(x)`.

Возвращает строку с именем типа. Например, `"string"`.

Для `null` возвращается `"object"` – это ошибка в языке, на самом деле это не объект.

В следующих главах мы сконцентрируемся на примитивных значениях, а когда познакомимся с ними, перейдём к объектам.

Задачи

Шаблонные строки

важность: 5

Что выведет этот скрипт?

```
let name = "Ilya";

alert( `hello ${1}` ); // ?

alert( `hello ${"name"}` ); // ?

alert( `hello ${name}` ); // ?
```

К решению

Взаимодействие: alert, prompt, confirm

Так как мы будем использовать браузер как демо-среду, нам нужно познакомиться с несколькими функциями его интерфейса, а именно: `alert`, `prompt` и `confirm`.

alert

С этой функцией мы уже знакомы. Она показывает сообщение и ждёт, пока пользователь нажмёт кнопку «ОК».

Например:

```
alert("Hello");
```

Это небольшое окно с сообщением называется *модальным окном*. Понятие *модальное* означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

prompt

Функция `prompt` принимает два аргумента:

```
result = prompt(title, [default]);
```

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

`title`

Текст для отображения в окне.

`default`

Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

> **Квадратные скобки в синтаксисе `[...]`**
>
> Квадратные скобки вокруг `default` в описанном выше синтаксисе означают, что параметр факультативный, необязательный.

Пользователь может напечатать что-либо в поле ввода и

нажать OK. Введённый текст будет присвоен переменной `result`. Пользователь также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу `Esc`. В этом случае значением `result` станет `null`.

Вызов `prompt` возвращает текст, указанный в поле для ввода, или `null`, если ввод отменён пользователем.

Например:

```
let age = prompt('Сколько тебе лет?', 100);

alert(`Тебе ${age} лет!`); // Тебе 100 лет!
```

Для IE: всегда устанавливайте значение по умолчанию

Второй параметр является необязательным, но если не указать его, то Internet Explorer вставит строку `"undefined"` в поле для ввода.

Запустите код в Internet Explorer и посмотрите на результат:

```
let test = prompt("Test");
```

Чтобы `prompt` хорошо выглядел в IE, рекомендуется всегда указывать второй параметр:

```
let test = prompt("Test", ''); // <-- для IE
```

confirm

Синтаксис:

```
result = confirm(question);
```

Функция `confirm` отображает модальное окно с текстом вопроса `question` и двумя кнопками: OK и Отмена.

Результат – `true`, если нажата кнопка OK. В других случаях – `false`.

Например:

```
let isBoss = confirm("Ты здесь главный?");

alert( isBoss ); // true, если нажата OK
```

Итого

Мы рассмотрели 3 функции браузера для взаимодействия с пользователем:

`alert`

показывает сообщение.

`prompt`

показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный в поле ввода текст или `null`, если была нажата кнопка «Отмена» или `Esc` с клавиатуры.

`confirm`

показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена. Возвращает `true`, если нажата OK, и `false`, если нажата кнопка «Отмена» или `Esc` с клавиатуры.

Все эти методы являются модальными: останавливают

выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

Такова цена простоты. Есть другие способы показать более приятные глазу окна с богатой функциональностью для взаимодействия с пользователем, но если «навороты» не имеют значения, то данные методы работают отлично.

Задачи

Простая страница

важность: 4

Создайте страницу, которая спрашивает имя у пользователя и выводит его.

Запустить демо

К решению

Преобразование типов

Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу.

Например, `alert` автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам.

Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип.

> **Пока что мы не говорим об объектах**
>
> В этой главе мы не касаемся объектов. Сначала мы разберём преобразование примитивных значений. Мы разберём преобразование объектов позже, в главе Преобразование объектов в примитивы.

Строковое преобразование

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.

Например, `alert(value)` преобразует значение к строке.

Также мы можем использовать функцию `String(value)`, чтобы преобразовать значение к строке:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // теперь value это строка "true"
alert(typeof value); // string
```

Преобразование происходит очевидным образом. `false`

становится `"false"`, `null` становится `"null"` и т.п.

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.

Например, когда операция деления `/` применяется не к числу:

```
alert( "6" / "2" ); // 3, строки преобразуются в числа
```

Мы можем использовать функцию `Number(value)`, чтобы явно преобразовать `value` к числу:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number
```

Явное преобразование часто применяется, когда мы ожидаем получить число из строкового контекста, например из текстовых полей форм.

Если строка не может быть явно приведена к числу, то результатом преобразования будет `NaN`. Например:

```
let age = Number("Любая строка вместо числа");

alert(age); // NaN, преобразование не удалось
```

Правила численного преобразования:

Значение	Преобразуется в…

`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем `0`, иначе из непустой строки «считывается» число. При ошибке результат `NaN`.

Примеры:

```
alert( Number("   123   ") ); // 123
alert( Number("123z") );      // NaN (ошибка чтения числа на месте символа "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Учтите, что `null` и `undefined` ведут себя по-разному. Так, `null` становится нулём, тогда как `undefined` приводится к `NaN`.

Большинство математических операторов также производит данное преобразование, как мы увидим в следующей главе.

Логическое преобразование

Логическое преобразование самое простое.

Происходит в логических операциях (позже мы познакомимся с условными проверками и подобными конструкциями), но также может быть выполнено явно с помощью функции `Boolean(value)`.

Правило преобразования:

> Значения, которые интуитивно «пустые», вроде `0`, пустой строки, `null`, `undefined` и `NaN`, становятся `false`.
>
> Все остальные значения становятся `true`.

Например:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("Привет!") ); // true
alert( Boolean("") ); // false
```

> **Заметим, что строчка с нулём `"0"` — это `true`**
>
> Некоторые языки (к примеру, PHP) воспринимают строку `"0"` как `false`. Но в JavaScript, если строка не пустая, то она всегда `true`.
>
> ```
> alert( Boolean("0") ); // true
> alert( Boolean(" ") ); // пробел это тоже true
> (любая непустая строка это true)
> ```

Итого

Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое.

`Строковое` – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью `String(value)`. Для примитивных значений работает очевидным образом.

`Численное` – Происходит в математических операциях.

Может быть вызвано с помощью `Number(value)`.

Преобразование подчиняется правилам:

Значение	Становится…
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем `0`, иначе из непустой строки «считывается» число. При ошибке результат `NaN`.

Логическое – Происходит в логических операциях. Может быть вызвано с помощью `Boolean(value)`.

Подчиняется правилам:

Значение	Становится…
`0`, `null`, `undefined`, `NaN`, `""`	`false`
любое другое значение	`true`

Большую часть из этих правил легко понять и запомнить. Особые случаи, в которых часто допускаются ошибки:

`undefined` при численном преобразовании становится `NaN`, не `0`.

`"0"` и строки из одних пробелов типа `" "` при логическом преобразовании всегда `true`.

В этой главе мы не говорили об объектах. Мы вернёмся к ним позже, в главе Преобразование объектов в примитивы, посвящённой только объектам, сразу после того, как узнаем больше про основы JavaScript.

Базовые операторы, математика

Многие операторы знакомы нам ещё со школы: сложение `+`, умножение `*`, вычитание `-` и так далее.

В этой главе мы начнём с простых операторов, а потом сконцентрируемся на специфических для JavaScript аспектах, которые не проходят в школьном курсе арифметики.

Термины: «унарный», «бинарный», «операнд»

Прежде, чем мы двинемся дальше, давайте разберёмся с терминологией.

Операнд – то, к чему применяется оператор. Например, в умножении `5 * 2` есть два операнда: левый операнд равен `5`, а правый операнд равен `2`. Иногда их называют «аргументами» вместо «операндов».

Унарным называется оператор, который применяется к одному операнду. Например, оператор унарный минус `"-"` меняет знак числа на противоположный:

```
let x = 1;

alert( x ); // -1, применили унарный минус
x = -x;
```

Бинарным называется оператор, который применяется

к двум операндам. Тот же минус существует и в бинарной форме:

```
let x = 1, y = 3;
alert( y - x ); // 2, бинарный минус вычитает значения
```

Формально, в последних примерах мы говорим о двух разных операторах, использующих один символ: оператор отрицания (унарный оператор, который обращает знак) и оператор вычитания (бинарный оператор, который вычитает одно число из другого).

Математика

Поддерживаются следующие математические операторы:

Сложение `+`,

Вычитание `-`,

Умножение `*`,

Деление `/`,

Взятие остатка от деления `%`,

Возведение в степень `**`.

Первые четыре оператора очевидны, а про `%` и `**` стоит сказать несколько слов.

Взятие остатка %

Оператор взятия остатка `%`, несмотря на обозначение, никакого отношения к процентам не имеет.

Результат `a % b` – это остаток от целочисленного деления `a` на `b`.

Например:

```
alert( 5 % 2 ); // 1, остаток от деления 5 на 2
```

```
alert( 8 % 3 ); // 2, остаток от деления 8 на 3
```

Возведение в степень **

В выражении `a ** b` оператор возведения в степень умножает `a` на само себя `b` раз.

Например:

```
alert( 2 ** 2 ); // 4  (2 умножено на себя 2 раза)
alert( 2 ** 3 ); // 8  (2 * 2 * 2, 3 раза)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2, 4 раза)
```

Математически, оператор работает и для нецелых чисел. Например, квадратный корень является возведением в степень `1/2`:

```
alert( 4 ** (1/2) ); // 2 (степень 1/2 эквивалентна взятию квадратного корня)
alert( 8 ** (1/3) ); // 2 (степень 1/3 эквивалентна взятию кубического корня)
```

Сложение строк при помощи бинарного +

Давайте рассмотрим специальные возможности операторов JavaScript, которые выходят за рамки школьной арифметики.

Обычно при помощи плюса `'+'` складывают числа.

Но если бинарный оператор `'+'` применить к строкам, то он их объединяет в одну:

```
let s = "моя" + "строка";
alert(s); // моястрока
```

Обратите внимание, если хотя бы один операнд является строкой, то второй будет также преобразован в строку.

Например:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Как видите, не важно, первый или второй операнд является строкой.

Вот пример посложнее:

```
alert(2 + 2 + '1' ); // будет "41", а не "221"
```

Здесь операторы работают один за другим. Первый `+` складывает два числа и возвращает `4`, затем следующий `+` объединяет результат со строкой, производя действие `4 + '1' = 41`.

Сложение и преобразование строк — это особенность бинарного плюса `+`. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа.

Например, вычитание и деление:

```
alert( 6 - '2' ); // 4, '2' приводится к числу
alert( '6' / '2' ); // 3, оба операнда приводятся к числам
```

Приведение к числу, унарный +

Плюс `+` существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть применённый к одному значению, плюс `+` ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число.

Например:

```
// Не влияет на числа
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Преобразует не числа в числа
alert( +true ); // 1
alert( +"" );   // 0
```

На самом деле это то же самое, что и `Number(...)`, только короче.

Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы — это строки. А что, если их нужно, к примеру, сложить?

Бинарный плюс сложит их как строки:

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // "23", так как бинарный плюс объединяет строки
```

Поэтому используем унарный плюс, чтобы преобразовать к числу:

```
let apples = "2";
let oranges = "3";



// более длинный вариант
// alert( Number(apples) + Number(oranges) ); // 5
// оба операнда предварительно преобразованы в числа
alert( +apples + +oranges ); // 5
```

С точки зрения математика, такое изобилие плюсов

выглядит странным. Но с точки зрения программиста тут нет ничего особенного: сначала выполнятся унарные плюсы, которые приведут строки к числам, а затем бинарный `'+'` их сложит.

Почему унарные плюсы выполнились до бинарного сложения? Как мы сейчас увидим, дело в их приоритете.

Приоритет операторов

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*, или, другими словами, существует определённый порядок выполнения операторов.

Из школы мы знаем, что умножение в выражении `1 + 2 * 2` выполнится раньше сложения. Это как раз и есть «приоритет». Говорят, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что, если мы не удовлетворены порядком по умолчанию, мы можем использовать их, чтобы изменить приоритет. Например, написать `(1 + 2) * 2`.

В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше, – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Отрывок из таблицы приоритетов (нет необходимости всё запоминать, обратите внимание, что приоритет унарных операторов выше, чем соответствующих бинарных):

Приоритет	Название	Обозначение
…	…	…
17	унарный плюс	+

17	унарный минус	−
16	возведение в степень	**
15	умножение	*
15	деление	/
13	сложение	+
13	вычитание	−
…	…	…
3	присваивание	=
…	…	…

Так как «унарный плюс» имеет приоритет `17`, который выше, чем `13` у «сложения» (бинарный плюс), то в выражении `"+apples + +oranges"` сначала выполнятся унарные плюсы, а затем сложение.

Присваивание

Давайте отметим, что в таблице приоритетов также есть оператор присваивания `=`. У него один из самых низких приоритетов: `3`.

Именно поэтому, когда переменной что-либо присваивают, например, `x = 2 * 2 + 1`, то сначала выполнится арифметика, а уже затем произойдёт присваивание `=` с сохранением результата в `x`.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

Присваивание = возвращает значение

Тот факт, что `=` является оператором, а не «магической» конструкцией языка, имеет интересные последствия.

Большинство операторов в JavaScript возвращают значение. Для некоторых это очевидно, например сложение `+` или умножение `*`. Но и оператор присваивания не является исключением.

Вызов `x = value` записывает `value` в `x` *и возвращает его*.

Благодаря этому присваивание можно использовать как часть более сложного выражения:

```
let a = 1;
let b = 2;
```

```
alert( a ); // 3
alert( c ); // 0
let c = 3 - (a = b + 1);
```

В примере выше результатом `(a = b + 1)` будет значение, которое присваивается переменной `a` (то есть `3`). Потом оно используется для дальнейших вычислений.

Забавное применение присваивания, не так ли? Нам нужно понимать, как это работает, потому что иногда это можно увидеть в JavaScript-библиотеках.

Однако писать самим в таком стиле не рекомендуется. Такие трюки не сделают ваш код более понятным или читабельным.

Присваивание по цепочке

Рассмотрим ещё одну интересную возможность: цепочку присваиваний.

```
let a, b, c;
```

```
alert( a ); // 4
alert( b ); // 4
```

```
alert( c ); // 4
a = b = c = 2 + 2;
```

Такое присваивание работает справа налево. Сначала вычисляется самое правое выражение `2 + 2`, и затем результат присваивается переменным слева: `c`, `b` и `a`. В конце у всех переменных будет одно значение.

Опять-таки, чтобы код читался легче, лучше разделять подобные конструкции на несколько строчек:

```
c = 2 + 2;
b = c;
a = c;
```

Польза от такого стиля особенно ощущается при быстром просмотре кода.

Сокращённая арифметика с присваиванием

Часто нужно применить оператор к переменной и сохранить результат в ней же.

Например:

```
let n = 2;
n = n + 5;
n = n * 2;
```

Эту запись можно укоротить при помощи совмещённых операторов `+=` и `*=`:

```
let n = 2;
n += 5; // теперь n = 7 (работает как n = n + 5)
n *= 2; // теперь n = 14 (работает как n = n * 2)

alert( n ); // 14
```

Подобные краткие формы записи существуют для всех арифметических и побитовых операторов: `/=`, `-=` и так далее.

Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

```
let n = 2;

n *= 3 + 5;

alert( n ); // 16  (сначала выполнится правая часть, выражение идентично n *= 8)
```

Инкремент/декремент

Одной из наиболее частых числовых операций является увеличение или уменьшение на единицу.

Для этого существуют даже специальные операторы:

Инкремент ++ увеличивает переменную на 1:

```
let counter = 2;
counter++;        // работает как counter = counter + 1, просто запись короче
alert( counter ); // 3
```

Декремент -- уменьшает переменную на 1:

```
let counter = 2;
counter--;        // работает как counter = counter - 1, просто запись короче
alert( counter ); // 1
```

Важно:

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, приведёт к ошибке.

Операторы `++` и `--` могут быть расположены не только после, но и до переменной.

Когда оператор идёт после переменной — это «постфиксная форма»: `counter++`.

«Префиксная форма» — это когда оператор идёт перед переменной: `++counter`.

Обе эти инструкции делают одно и то же: увеличивают `counter` на `1`.

Есть ли разница между ними? Да, но увидеть её мы сможем, только если будем использовать значение, которое возвращают `++/--`.

Давайте проясним этот момент. Как мы знаем, все операторы возвращают значение. Операторы инкремента/декремента не исключение. Префиксная форма возвращает новое значение, в то время как постфиксная форма возвращает старое (до увеличения/уменьшения числа).

Чтобы увидеть разницу, вот небольшой пример:

```
let counter = 1;
let a = ++counter; // (*)

alert(a); //
2
```

В строке `(*)` *префиксная* форма `++counter` увеличивает `counter` и возвращает новое значение `2`. Так что `alert` покажет `2`.

Теперь посмотрим на постфиксную форму:

```
let counter = 1;
let a = counter++; // (*) меняем ++counter на counter++

alert(a); //
1
```

В строке `(*)` *постфиксная* форма `counter++` также увеличивает `counter`, но возвращает *старое* значение (которое было до увеличения). Так что `alert` покажет `1`.

Подведём итоги:

Если результат оператора не используется, а нужно только увеличить/уменьшить переменную, тогда без разницы, какую форму использовать:

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, обе строки сделали одно и то же
```

Если хочется тут же использовать результат, то нужна префиксная форма:

```
let counter = 0;
alert( ++counter ); // 1
```

Если нужно увеличить и при этом получить значение переменной *до увеличения* – нужна постфиксная форма:

```
let counter = 0;
alert( counter++ ); // 0
```

Инкремент/декремент можно использовать в любых выражениях

Операторы `++/--` могут также использоваться внутри выражений. Их приоритет выше, чем у большинства других арифметических операций.

Например:

```
let counter = 1;
alert( 2 * ++counter ); // 4
```

Сравните с:

```
let counter = 1;
alert( 2 * counter++ ); // 2, потому что counter++
возвращает "старое" значение
```

Хотя технически здесь всё в порядке, такая запись обычно делает код менее читабельным. Одна строка выполняет множество действий – нехорошо.

При беглом чтении кода можно с лёгкостью пропустить такой `counter++`, и будет неочевидно, что переменная увеличивается.

Лучше использовать стиль «одна строка – одно действие»:

```
let counter = 1;
alert( 2 * counter );
counter++;
```

Побитовые операторы

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления.

Эти операторы не являются чем-то специфичным для

JavaScript, они поддерживаются в большинстве языков программирования.

Поддерживаются следующие побитовые операторы:

- AND(и) (`&`)
- OR(или) (`|`)
- XOR(побитовое исключающее или) (`^`)
- NOT(не) (`~`)
- LEFT SHIFT(левый сдвиг) (`<<`)
- RIGHT SHIFT(правый сдвиг) (`>>`)
- ZERO-FILL RIGHT SHIFT(правый сдвиг с заполнением нулями) (`>>>`)

Они используются редко, когда возникает необходимость оперировать с числами на очень низком (побитовом) уровне. В ближайшем времени они нам не понадобятся, так как веб-разработчики редко к ним прибегают, хотя в некоторых сферах (например, в криптографии) они полезны. Можете прочитать раздел о них на MDN, когда возникнет реальная необходимость.

Оператор «запятая»

Оператор «запятая» (`,`) редко применяется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нам нужно знать его, чтобы понимать, что при этом происходит.

Оператор «запятая» предоставляет нам возможность вычислять несколько выражений, разделяя их запятой `,`. Каждое выражение выполняется, но возвращается результат только последнего.

Например:

```
alert( a ); // 7 (результат вычисления 3 + 4)
let a = (1 + 2, 3 + 4);
```

Первое выражение `1 + 2` выполняется, а результат отбрасывается. Затем идёт `3 + 4`, выражение выполняется и возвращается результат.

Запятая имеет очень низкий приоритет

Пожалуйста, обратите внимание, что оператор `,` имеет очень низкий приоритет, ниже `=`, поэтому скобки важны в приведённом выше примере.

Без них в `a = 1 + 2, 3 + 4` сначала выполнится `+`, суммируя числа в `a = 3, 7`, затем оператор присваивания `=` присвоит `a = 3`, а то, что идёт дальше, будет игнорировано. Всё так же, как в `(a = 1 + 2), 3 + 4`.

Зачем нам оператор, который отбрасывает всё, кроме последнего выражения?

Иногда его используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке.

Например:

```
// три операции в одной строке
for (; a < 10; a++) {
  ...
}
a = 1, b = 3, c = a * b
```

Такие трюки используются во многих JavaScript-

фреймворках. Вот почему мы упоминаем их. Но обычно они не улучшают читабельность кода, поэтому стоит хорошо подумать, прежде чем их использовать.

Задачи

Постфиксная и префиксная формы

важность: 5

Чему будут равны переменные `a`, `b`, `c` и `d` в примере ниже?

```
let a = 1, b = 1;

let c = ++a; // ?
let d = b++; // ?
```

К решению

Результат присваивания

важность: 3

Чему будут равны переменные `a` и `x` после исполнения кода в примере ниже?

```
let a = 2;

let x = 1 + (a *= 2);
```

К решению

Преобразование типов

важность: 5

Какой результат будет у выражений ниже?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
7 / 0
"  -9  " + 5
"  -9  " - 5
null + 1
undefined + 1
" \t \n" - 2
```

Подумайте как следует, запишите ответы и сверьтесь с решением.

К решению

Исправьте сложение

важность: 5

Ниже приведён код, который запрашивает у пользователя два числа и показывает их сумму.

Он работает неправильно. Код в примере выводит `12` (для значения полей по умолчанию).

В чём ошибка? Исправьте её. Результат должен быть `3`.

```
let a = prompt("Первое число?", 1);
let b = prompt("Второе число?", 2);

alert(a + b); // 12
```

К решению

Операторы сравнения

Многие операторы сравнения известны нам из математики.

В JavaScript они записываются так:

- Больше/меньше: `a > b`, `a < b`.
- Больше/меньше или равно: `a >= b`, `a <= b`.
- Равно: `a == b`. Обратите внимание, для сравнения используется двойной знак равенства `==`. Один знак равенства `a = b` означал бы присваивание.
- Не равно. В математике обозначается символом $\neq$, но в JavaScript записывается как `a != b`.

В этом разделе мы больше узнаем про то, какие бывают сравнения, как язык с ними работает и к каким неожиданностям мы должны быть готовы.

В конце вы найдёте хороший рецепт того, как избегать «причуд» сравнения в JavaScript.

Результат сравнения имеет логический тип

Все операторы сравнения возвращают значение логического типа:

- `true` – означает «да», «верно», «истина».
- `false` – означает «нет», «неверно», «ложь».

Например:

```
alert( 2 > 1 );  // true (верно)
alert( 2 == 1 ); // false (неверно)
alert( 2 != 1 ); // true (верно)
```

Результат сравнения можно присвоить переменной, как и любое значение:

```
let result = 5 > 4; // результат сравнения присваивается переменной result
alert( result ); // true
```

Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок.

Другими словами, строки сравниваются посимвольно.

Например:

```
alert( 'Я' > 'А' ); // true
alert( 'Коты' > 'Кода' ); // true
alert( 'Сонный' > 'Сон' ); // true
```

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. Сравнение завершено.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

В примерах выше сравнение `'Я' > 'A'` завершится на первом шаге, тогда как строки `'Коты'` и `'Кода'` будут сравниваться посимвольно:

1. `К` равна `К`.
2. `о` равна `о`.
3. `т` больше, чем `д`. На этом сравнение заканчивается. Первая строка больше.

Используется кодировка Unicode, а не настоящий алфавит

Приведённый выше алгоритм сравнения похож на алгоритм, используемый в словарях и телефонных книгах, но между ними есть и различия.

Например, в JavaScript имеет значение регистр символов. Заглавная буква `"A"` не равна строчной `"a"`. Какая же из них больше? Строчная `"a"`. Почему? Потому что строчные буквы имеют больший код во внутренней таблице кодирования, которую использует JavaScript (Unicode). Мы ещё поговорим о внутреннем представлении строк и его влиянии в главе Строки.

Сравнение разных типов

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

Например:

```
alert( '2' > 1 ); // true, строка '2' становится числом 2
```

```
alert( '01' == 1 ); // true, строка '01' становится числом 1
```

Логическое значение `true` становится `1`, а `false` – `0`.

Например:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Забавное следствие

Возможна следующая ситуация:

> Два значения равны.
>
> Одно из них `true` как логическое значение, другое – `false`.

Например:

```
let a = 0;
alert( Boolean(a) ); // false

let b = "0";
alert( Boolean(b) ); // true

alert(a == b); // true!
```

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому `"0"` становится `0`. В то время как явное преобразование с помощью `Boolean` использует другой набор правил.

Строгое сравнение

Использование обычного сравнения `==` может вызывать проблемы. Например, оно не отличает `0` от `false`:

```
alert( 0 == false ); // true
```

Та же проблема с пустой строкой:

```
alert( '' == false ); // true
```

Это происходит из-за того, что операнды разных типов преобразуются оператором `==` к числу. В итоге, и пустая строка, и `false` становятся нулём.

Как же тогда отличать `0` от `false`?

Оператор строгого равенства `===` проверяет равенство без приведения типов.

Другими словами, если `a` и `b` имеют разные типы, то проверка `a === b` немедленно возвращает `false` без попытки их преобразования.

Давайте проверим:

```
alert( 0 === false ); // false, так как сравниваются разные типы
```

Ещё есть оператор строгого неравенства `!==`, аналогичный `!=`.

Оператор строгого равенства дольше писать, но он делает код более очевидным и оставляет меньше места для ошибок.

Сравнение с null и undefined

Поведение `null` и `undefined` при сравнении с другими

значениями — особое:

При строгом равенстве `===`

Эти значения различны, так как различны их типы.

```
alert( null === undefined ); // false
```

При нестрогом равенстве `==`

Эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

```
alert( null == undefined ); // true
```

При использовании математических операторов и других операторов сравнения `< > <= >=`

Значения `null/undefined` преобразуются к числам: `null` становится `0`, а `undefined` – `NaN`.

Посмотрим, какие забавные вещи случаются, когда мы применяем эти правила. И, что более важно, как избежать ошибок при их использовании.

Странный результат сравнения null и 0

Сравним `null` с нулём:

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3)
true
```

С точки зрения математики это странно. Результат последнего сравнения говорит о том, что "`null` больше или равно нулю", тогда результат одного из сравнений выше должен быть `true`, но они оба ложны.

Причина в том, что нестрогое равенство и сравнения `> <`

`>=` `<=` работают по-разному. Сравнения преобразуют `null` в число, рассматривая его как `0`. Поэтому выражение (3) `null >= 0` истинно, а `null > 0` ложно.

С другой стороны, для нестрогого равенства `==` значений `undefined` и `null` действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) `null == 0` ложно.

Несравненное значение undefined

Значение `undefined` несравнимо с другими значениями:

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

Почему же сравнение `undefined` с нулём всегда ложно?

На это есть следующие причины:

Сравнения `(1)` и `(2)` возвращают `false`, потому что `undefined` преобразуется в `NaN`, а `NaN` – это специальное числовое значение, которое возвращает `false` при любых сравнениях.

Нестрогое равенство `(3)` возвращает `false`, потому что `undefined` равно только `null`, `undefined` и ничему больше.

Как избежать проблем

Зачем мы рассмотрели все эти примеры? Должны ли мы постоянно помнить обо всех этих особенностях? Не обязательно. Со временем все они станут вам знакомы, но можно избежать проблем, если следовать надёжным правилам:

Относитесь очень осторожно к любому сравнению с `undefined/null`, кроме случаев строгого равенства `===`.

Не используйте сравнения `>= > < <=` с переменными, которые могут принимать значения `null/undefined`, разве что вы полностью уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

Итого

Операторы сравнения возвращают значения логического типа.

Строки сравниваются посимвольно в лексикографическом порядке.

Значения разных типов при сравнении приводятся к числу. Исключением является сравнение с помощью операторов строгого равенства/неравенства.

Значения `null` и `undefined` равны `==` друг другу и не равны любому другому значению.

Будьте осторожны при использовании операторов сравнений вроде `>` и `<` с переменными, которые могут принимать значения `null/undefined`. Хорошей идеей будет сделать отдельную проверку на `null/undefined`.

Задачи

Операторы сравнения

важность: 5

Каким будет результат этих выражений?

```
5 > 4
"ананас" > "яблоко"
"2" > "12"
undefined == null
undefined === null
null == "\n0\n"
null === +"\n0\n"
```

К решению

Условное ветвление: if, '?'

Иногда нам нужно выполнить различные действия в зависимости от условий.

Для этого мы можем использовать инструкцию `if` и условный оператор `?`, который также называют оператором «вопросительный знак».

Инструкция «if»

Инструкция `if(...)` вычисляет условие в скобках и, если результат `true`, то выполняет блок кода.

Например:

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');

if (year == 2015) alert( 'Вы правы!' );
```

В примере выше, условие – это простая проверка на равенство (`year == 2015`), но оно может быть и гораздо более сложным.

Если мы хотим выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки:

```
if (year == 2015) {
  alert( "Правильно!" );
  alert( "Вы такой умный!" );
}
```

Мы рекомендуем использовать фигурные скобки `{}` всегда, когда вы используете инструкцию `if`, даже если выполняется только одна команда. Это улучшает читабельность кода.

Преобразование к логическому типу

Инструкция `if (…)` вычисляет выражение в скобках и преобразует результат к логическому типу.

Давайте вспомним правила преобразования типов из главы Преобразование типов:

> Число `0`, пустая строка `""`, `null`, `undefined` и `NaN` становятся `false`. Из-за этого их называют «ложными» («falsy») значениями.
>
> Остальные значения становятся `true`, поэтому их называют «правдивыми» («truthy»).

Таким образом, код при таком условии никогда не выполнится:

```
if (0) { // 0 is falsy
  ...
}
```

…а при таком – выполнится всегда:

```
if (1) { // 1 is truthy
  ...
}
```

Мы также можем передать заранее вычисленное в переменной логическое значение в `if`, например так:

```
let condition = (year == 2015); // преобразуется к true или
false

if (condition) {
  ...
}
```

Блок «else»

Инструкция `if` может содержать необязательный блок «else» («иначе»). Он выполняется, когда условие ложно.

Например:

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');

if (year == 2015) {
  alert( 'Да вы знаток!' );
} else {
  alert( 'А вот и неправильно!' ); // любое значение, кроме 2015
}
```

Несколько условий: «else if»

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок `else if`.

Например:

```
let year = prompt('В каком году была опубликована спецификация ECMAScript-2015?', '');

if (year < 2015) {
  alert( 'Это слишком рано...' );
} else if (year > 2015) {
  alert( 'Это поздновато' );
} else {
  alert( 'Верно!' );
}
```

В приведённом выше коде JavaScript сначала проверит `year < 2015`. Если это неверно, он переходит к следующему условию `year > 2015`. Если оно тоже ложно, тогда сработает последний `alert`.

Блоков `else if` может быть и больше. Присутствие блока `else` не является обязательным.

Условный оператор „?“

Иногда нам нужно определить переменную в зависимости от условия.

Например:

```
let accessAllowed;
let age = prompt('Сколько вам лет?', '');


alert(accessAllowed);
if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
```

Так называемый «условный» оператор «вопросительный знак» позволяет нам сделать это более коротким и простым способом.

Оператор представлен знаком вопроса `?`. Его также называют «тернарный», так как этот оператор, единственный в своём роде, имеет три аргумента.

Синтаксис:

```
let result = условие ? значение1 : значение2;
```

Сначала вычисляется `условие`: если оно истинно, тогда возвращается `значение1`, в противном случае – `значение2`.

Например:

```
let accessAllowed = (age > 18) ? true : false;
```

Технически, мы можем опустить круглые скобки вокруг `age > 18`. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения `>`.

Этот пример будет делать то же самое, что и предыдущий:

```
// оператор сравнения "age > 18" выполняется первым в любом случае
// (нет необходимости заключать его в скобки)
let accessAllowed = age > 18 ? true : false;
```

Но скобки делают код более читабельным, поэтому мы рекомендуем их использовать.

На заметку:

В примере выше вы можете избежать использования оператора вопросительного знака `?`, т.к. сравнение само по себе уже возвращает `true/false`:

```
// то же самое
let accessAllowed = age > 18;
```

Несколько операторов „?“

Последовательность операторов вопросительного знака `?` позволяет вернуть значение, которое зависит от более чем одного условия.

Например:

```
let age = prompt('Возраст?', 18);

let message = (age < 3) ? 'Здравствуй, малыш!' :
  (age < 18) ? 'Привет!' :
  (age < 100) ? 'Здравствуйте!' :
```

```
  'Какой необычный возраст!';

alert( message );
```

Поначалу может быть сложно понять, что происходит. Но при ближайшем рассмотрении мы видим, что это обычная последовательная проверка:

1. Первый знак вопроса проверяет `age < 3`.
2. Если верно – возвращает `'Здравствуй, малыш!'`. В противном случае, проверяет выражение после двоеточия „":"“, вычисляет `age < 18`.
3. Если это верно – возвращает `'Привет!'`. В противном случае, проверяет выражение после следующего двоеточия „":"“, вычисляет `age < 100`.
4. Если это верно – возвращает `'Здравствуйте!'`. В противном случае, возвращает выражение после последнего двоеточия – `'Какой необычный возраст!'`.

Вот как это выглядит при использовании `if..else`:

```
if (age < 3) {
  message = 'Здравствуй, малыш!';
} else if (age < 18) {
  message = 'Привет!';
} else if (age < 100) {
  message = 'Здравствуйте!';
} else {
  message = 'Какой необычный возраст!';
}
```

Нетрадиционное использование „?“

Иногда оператор «вопросительный знак» `?` используется в качестве замены `if`:

```
let company = prompt('Какая компания создала JavaScript?',
'');
```

```
(company == 'Netscape') ?
   alert('Верно!') : alert('Неправильно.');
```

В зависимости от условия `company == 'Netscape'`, будет выполнена либо первая, либо вторая часть после `?`.

Здесь мы не присваиваем результат переменной. Вместо этого мы выполняем различный код в зависимости от условия.

Не рекомендуется использовать оператор вопросительного знака таким образом.

Несмотря на то, что такая запись короче, чем эквивалентная инструкция `if`, она менее читабельна.

Вот, для сравнения, тот же код, использующий `if`:

```
let company = prompt('Какая компания создала JavaScript?',
'');

if (company == 'Netscape') {
  alert('Верно!');
} else {
  alert('Неправильно.');
}
```

При чтении глаза сканируют код по вертикали. Блоки кода, занимающие несколько строк, воспринимаются гораздо легче, чем длинный горизонтальный набор инструкций.

Смысл оператора «вопросительный знак» `?` – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его именно для этого. Когда вам нужно выполнить разные ветви кода – используйте `if`.

Задачи

if (строка с нулём)

важность: 5

Выведется ли `alert`?

```
if ("0") {
  alert( 'Привет' );
}
```

К решению

Название JavaScript

важность: 2

Используя конструкцию `if..else`, напишите код, который будет спрашивать: „Какое «официальное» название JavaScript?“

Если пользователь вводит «ECMAScript», то показать: «Верно!», в противном случае – отобразить: «Не знаете? ECMAScript!»

Демо в новом окне

К решению

Покажите знак числа

важность: 2

Используя конструкцию `if..else`, напишите код, который получает число через `prompt`, а затем выводит в `alert`:

- `1`, если значение больше нуля,
- `-1`, если значение меньше нуля,
- `0`, если значение равно нулю.

Предполагается, что пользователь вводит только числа.

Демо в новом окне

К решению

Перепишите 'if' в '?'

важность: 5

Перепишите конструкцию `if` с использованием условного оператора `'?'`:

```
let result;

if (a + b < 4) {
  result = 'Мало';
} else {
  result = 'Много';
}
```

К решению

Перепишите 'if..else' в '?'

важность: 5

Перепишите `if..else` с использованием нескольких операторов `'?'`.

Для читаемости рекомендуется разбить код на несколько строк.

```
let message;

if (login == 'Сотрудник') {
  message = 'Привет';
} else if (login == 'Директор') {
  message = 'Здравствуйте';
} else if (login == '') {
  message = 'Нет логина';
} else {
  message = '';
}
```

К решению

Логические операторы

В JavaScript есть три логических оператора: `||` (ИЛИ), `&&` (И) и `!` (НЕ).

Несмотря на своё название, данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

Давайте рассмотрим их подробнее.

|| (ИЛИ)

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

```
result = a || b;
```

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов `true`, он вернёт `true`, в противоположной ситуации возвращается `false`.

В JavaScript, как мы увидим далее, этот оператор работает несколько иным образом. Но давайте спервa посмотрим, что происходит с булевыми значениями.

Существует всего четыре возможные логические комбинации:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

Как мы можем наблюдать, результат операций всегда равен `true`, за исключением случая, когда оба аргумента `false`.

Если значение не логического типа, то оно к нему приводится в целях вычислений.

Например, число `1` будет воспринято как `true`, а `0` – как `false`:

```
if (1 || 0) { // работает как if( true || false )
  alert( 'truthy!' );
}
```

Обычно оператор `||` используется в `if` для проверки

истинности любого из заданных условий.

К примеру:

```
let hour = 9;


  alert( 'Офис закрыт.' );
}
if (hour < 10 || hour > 18) {
```

Можно передать и больше условий:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'Офис закрыт.' ); // это выходной
}
```

ИЛИ «||» находит первое истинное значение

Описанная выше логика соответствует традиционной. Теперь давайте поработаем с «дополнительными» возможностями JavaScript.

Расширенный алгоритм работает следующим образом.

При выполнении ИЛИ || с несколькими значениями:

```
result = value1 || value2 || value3;
```

Оператор || выполняет следующие действия:

Вычисляет операнды слева направо.

Каждый операнд конвертирует в логическое значение. Если результат `true`, останавливается и возвращает исходное значение этого операнда.

Если все операнды являются ложными (`false`), возвращает последний из них.

Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ `"||"` возвращает первое истинное значение или последнее, если такое значение не найдено.

Например:

```
alert( 1 || 0 ); // 1
alert( true || 'no matter what' ); // true

alert( null || 1 ); // 1 (первое истинное значение)
alert( null || 0 || 1 ); // 1 (первое истинное значение)
alert( undefined || null || 0 ); // 0 (поскольку все ложно,
возвращается последнее значение)
```

Это делает возможным более интересное применение оператора по сравнению с «чистым, традиционным, только булевым ИЛИ».

1. **Получение первого истинного значения из списка переменных или выражений.**

   Представим, что у нас имеется ряд переменных, которые могут содержать данные или быть `null/undefined`. Как мы можем найти первую переменную с данными?

   С помощью `||`:

   ```
   let currentUser = null;
   let defaultUser = "John";


   alert( name ); // выбирается "John" – первое истинное
   значение
   let name = currentUser || defaultUser ||
   ```

```
"unnamed";
```

Если бы и `currentUser`, и `defaultUser` были ложными, в качестве результата мы бы наблюдали `"unnamed"`.

2. **Сокращённое вычисление.**

   Операндами могут быть как отдельные значения, так и произвольные выражения. ИЛИ вычисляет их слева направо. Вычисление останавливается при достижении первого истинного значения. Этот процесс называется «сокращённым вычислением», поскольку второй операнд вычисляется только в том случае, если первого недостаточно для вычисления всего выражения.

   Это хорошо заметно, когда выражение, указанное в качестве второго аргумента, имеет побочный эффект, например, изменение переменной.

   В приведённом ниже примере `x` не изменяется:

```
let x;

 || (x = 1);

alert(x); // undefined, потому что (x = 1) не вычисляется
true
```

   Если бы первый аргумент имел значение `false`, то `||` приступил бы к вычислению второго и выполнил операцию присваивания:

```
let x;

 || (x = 1);

alert(x); // 1
false
```

Присваивание – лишь один пример. Конечно, могут быть и другие побочные эффекты, которые не проявятся, если вычисление до них не дойдёт.

Как мы видим, этот вариант использования `||` является "аналогом `if`". Первый операнд преобразуется в логический. Если он оказывается ложным, начинается вычисление второго.

В большинстве случаев лучше использовать «обычный» `if`, чтобы облегчить понимание кода, но иногда это может быть удобно.

&& (И)

Оператор И пишется как два амперсанда `&&`:

```
result = a && b;
```

В традиционном программировании И возвращает `true`, если оба аргумента истинны, а иначе – `false`:

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

Пример с `if`:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'The time is 12:30' );
}
```

Как и в случае с ИЛИ, любое значение допускается в качестве операнда И:

```
if (1 && 0) { // вычисляется как true && false
  alert( "не сработает, так как результат ложный" );
}
```

И «&&» находит первое ложное значение

При нескольких подряд операторах И:

```
result = value1 && value2 && value3;
```

Оператор `&&` выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд преобразует в логическое значение. Если результат `false`, останавливается и возвращает исходное значение этого операнда.
- Если все операнды были истинными, возвращается последний.

Другими словами, И возвращает первое ложное значение. Или последнее, если ничего не найдено.

Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое *ложное* значение, а ИЛИ– первое *истинное*.

Примеры:

```
// Если первый операнд истинный,
// И возвращает второй:
alert( 1 && 0 ); // 0
alert( 1 && 5 ); // 5

// Если первый операнд ложный,
// И возвращает его. Второй операнд игнорируется
alert( null && 5 ); // null
alert( 0 && "no matter what" ); // 0
```

Можно передать несколько значений подряд. В таком

случае возвратится первое «ложное» значение, на котором остановились вычисления.

```
alert( 1 && 2 && null && 3 ); // null
```

Когда все значения верны, возвращается последнее

```
alert( 1 && 2 && 3 ); // 3
```

Приоритет оператора `&&` больше, чем у `||`

Приоритет оператора И `&&` больше, чем ИЛИ `||`, так что он выполняется раньше.

Таким образом, код `a && b || c && d` по существу такой же, как если бы выражения `&&` были в круглых скобках: `(a && b) || (c && d)`.

Как и оператор ИЛИ, И `&&` иногда может заменять `if`.

К примеру:

```
let x = 1;

(x > 0) && alert( 'Greater than zero!' );
```

Действие в правой части `&&` выполнится только в том случае, если до него дойдут вычисления. То есть, `alert` сработает, если в левой части `(x > 0)` будет `true`.

Получился аналог:

```
let x = 1;
```

```
if (x > 0) {
  alert( 'Greater than zero!' );
}
```

Однако, как правило, вариант с `if` лучше читается и воспринимается.

Он более очевиден, поэтому лучше использовать его.

! (НЕ)

Оператор НЕ представлен восклицательным знаком `!`.

Синтаксис довольно прост:

```
result = !value;
```

Оператор принимает один аргумент и выполняет следующие действия:

1. Сначала приводит аргумент к логическому типу `true/false`.
2. Затем возвращает противоположное значение.

Например:

```
alert( !true ); // false
alert( !0 ); // true
```

В частности, двойное НЕ используют для преобразования значений к логическому типу:

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
```

То есть первое НЕ преобразует значение в логическое значение и возвращает обратное, а второе НЕ снова инвертирует его. В конце мы имеем простое преобразование значения в логическое.

Есть немного более подробный способ сделать то же самое – встроенная функция `Boolean`:

```
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```

Приоритет НЕ `!` является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед `&&` или `||`.

Задачи

Что выведет alert (ИЛИ)?

важность: 5

Что выведет код ниже?

```
alert( null || 2 || undefined );
```

К решению

Что выведет alert (ИЛИ)?

важность: 3

Что выведет код ниже?

```
alert( alert(1) || 2 || alert(3) );
```

К решению

Что выведет alert (И)?

важность: 5

Что выведет код ниже?

```
alert( 1 && null && 2 );
```

К решению

Что выведет alert (И)?

важность: 3

Что выведет код ниже?

```
alert( alert(1) && alert(2) );
```

К решению

Что выведет этот код?

важность: 5

Что выведет код ниже?

```
alert( null || 2 && 3 || 4 );
```

К решению

Проверка значения из диапазона

важность: 3

Напишите условие `if` для проверки, что переменная `age` находится в диапазоне между `14` и `90` включительно.

«Включительно» означает, что значение переменной `age` может быть равно `14` или `90`.

К решению

Проверка значения вне диапазона

важность: 3

Напишите условие `if` для проверки, что значение переменной `age` НЕ находится в диапазоне `14` и `90` включительно.

Напишите два варианта: первый с использованием оператора НЕ `!`, второй – без этого оператора.

К решению

Вопрос о "if"

важность: 5

Какие из перечисленных ниже `alert` выполнятся?

Какие конкретно значения будут результатами выражений в условиях `if(...)`?

```
if (-1 || 0) alert( 'first' );
if (-1 && 0) alert( 'second' );
if (null || -1 && 1) alert( 'third' );
```

К решению

Проверка логина

важность: 3

Напишите код, который будет спрашивать логин с помощью `prompt`.

Если посетитель вводит «Админ», то `prompt` запрашивает пароль, если ничего не введено или нажата клавиша `Esc` – показать «Отменено», в противном случае отобразить «Я вас не знаю».

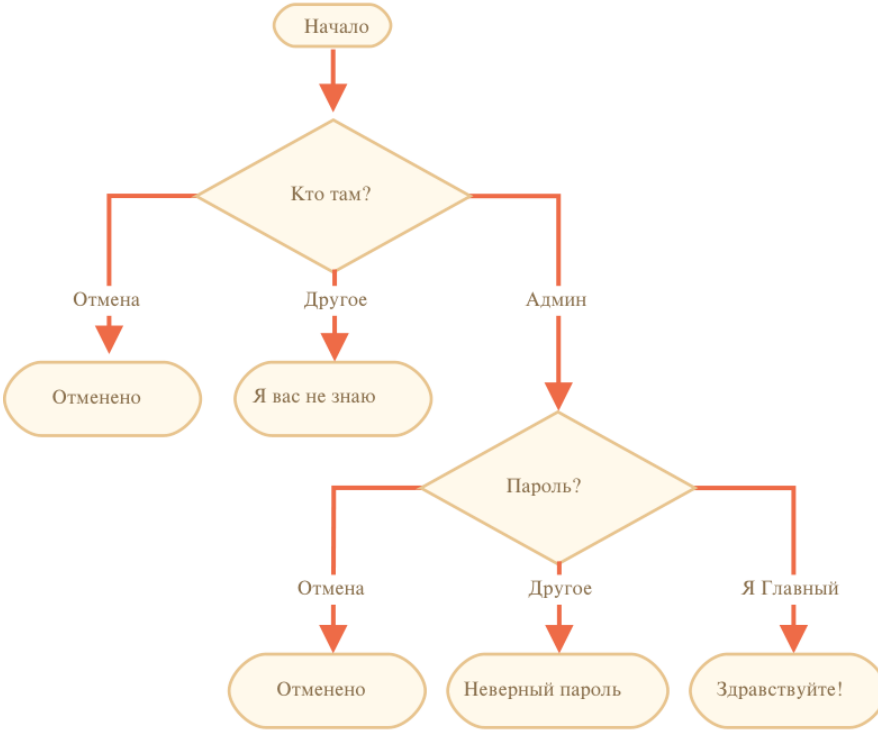
Пароль проверять так:

Если введён пароль «Я главный», то выводить «Здравствуйте!»,

Иначе – «Неверный пароль»,

При отмене – «Отменено».

Блок-схема:



Для решения используйте вложенные блоки `if`. Обращайте внимание на стиль и читаемость кода.

Подсказка: передача пустого ввода в приглашение `prompt` возвращает пустую строку `''`. Нажатие клавиши `Esc` во время запроса возвращает`null`.

Запустить демо

К решению

Оператор объединения с null '??'

Новая возможность

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

В этой статье мы будем говорить, что значение выражения «определено», если оно отличается от `null` или `undefined`.

Оператор объединения с null представляет собой два вопросительных знака `??`.

Результат выражения `a ?? b` будет следующим:

- `a`, если значение `a` определено,
- `b`, если значение `a` не определено.

То есть оператор `??` возвращает первый аргумент, если он не `null/undefined`, иначе второй.

Оператор объединения с null не является чем-то принципиально новым. Это всего лишь удобный синтаксис, как из двух значений получить одно «определённое».

Вот как можно переписать выражение `result = a ?? b`, используя уже знакомые нам операторы:

```
result = (a !== null && a !== undefined) ? a : b;
```

Как правило, оператор `??` нужен для того, чтобы задать значение по умолчанию для потенциально неопределённой переменной.

Например, в следующем примере, если переменная `user` не определена, покажем модальное окно с надписью `Аноним`:

```
let user;

alert(user ?? "Аноним"); // Аноним
```

Конечно, если бы переменная `user` содержала любое значение, кроме `null/undefined`, то мы бы увидели его:

```
let user = "Иван";

alert(user ?? "Аноним"); // Иван
```

Кроме этого, можно записать последовательность из операторов `??`, чтобы получить первое значение из списка, которое не является `null/undefined`.

Допустим, у нас есть данные пользователя в переменных `firstName`, `lastName` или `nickName`. Все они могут быть неопределёнными, если отсутствует соответствующая информация.

Выведем имя пользователя, используя одну из этих переменных, а в случае если все они не определены, то покажем «Аноним».

Для этого воспользуемся оператором `??`:

```
let firstName = null;
let lastName = null;
let nickName = "Суперкодер";

// показывает первое определённое значение:
alert(firstName ?? lastName ?? nickName ??
"Аноним"); // Суперкодер
```

Сравнение с ||

Оператор ИЛИ `||` можно использовать для того же, что и `??`, как это было показано в предыдущей главе.

Например, если в приведённом выше коде заменить `??` на `||`, то будет тот же самый результат:

```
let firstName = null;
let lastName = null;
let nickName = "Суперкодер";

// показывает первое истинное значение:
alert(firstName || lastName || nickName ||
"Аноним"); // Суперкодер
```

Оператор ИЛИ `||` существует с самого появления JavaScript, поэтому ранее для решения похожих задач разработчики использовали именно его.

С другой стороны, сравнительно недавно в язык был добавлен оператор объединения с null `??` как раз потому, что многие были недовольны оператором `||`.

Важное различие между ними заключается в том, что:

- `||` возвращает первое *истинное* значение.
- `??` возвращает первое *определённое* значение.

Проще говоря, оператор `||` не различает `false`, `0`, пустую строку `""` и `null/undefined`. Для него они все одинаковые, т.е. являются ложными значениями. Если первым аргументом для оператора `||` будет любое из перечисленных значений, то в качестве результата мы получим второй аргумент.

Однако на практике часто требуется использовать значение по умолчанию только тогда, когда переменная является `null/undefined`. Ведь именно тогда значение действительно неизвестно/не определено.

Например, рассмотрим следующий пример:

```
let height = 0;
```

```
alert(height || 100); // 100
alert(height ?? 100); // 0
```

`height || 100` проверяет, имеет ли переменная `height` ложное значение, что так и есть,

поэтому результатом является второй аргумент, т.е. `100`.

`height ?? 100` проверяет, что переменная `height` содержит `null/undefined`, а поскольку это не так,

то результатом является сама переменная `height`, т.е. `0`.

Если нулевая высота является «нормальным» значением, которое не должно заменяться значением по умолчанию, то оператор `??` делает как раз то, что нужно.

Приоритет

Оператор `??` имеет довольно низкий приоритет: `5`, согласно таблице на MDN. Таким образом, оператор `??` вычисляется до `=` и `?`, но после большинства других операций, таких как `+`, `*`.

Из этого следует, что если нужно выбрать значение при помощи оператора `??` вместе с другими операторами в выражении, следует добавить круглые скобки:

```
let height = null;
let width = null;

// важно: используйте круглые скобки
let area = (height ?? 100) * (width ?? 50);

alert(area); // 5000
```

Иначе, если опустить скобки, то оператор `*` выполнится первым, так как у него приоритет выше, чем у `??`, а это

приведёт к неправильным результатам.

```
// без круглых скобок
let area = height ?? 100 * width ?? 50;

// ...то же самое, что предыдущее выражение (вероятно, это не то, что нам нужно):
let area = height ?? (100 * width) ?? 50;
```

Использование ?? вместе с && или ||

По соображениям безопасности JavaScript запрещает использование оператора `??` вместе с `&&` и `||`, если только приоритет явно не указан в круглых скобках.

Выполнение следующего кода приведёт к синтаксической ошибке:

```
let x = 1 && 2 ?? 3; // Синтаксическая ошибка
```

Это довольно спорное ограничение, которое было описано в спецификации языка, чтобы избежать ошибок при замене оператора `||` на `??`.

Используйте круглые скобки, чтобы обойти это ограничение:

```
alert(x); // 2
let x = (1 && 2) ?? 3; // Работает без ошибок
```

Итого

Оператор объединения с null `??` — это быстрый способ выбрать первое «определённое» значение из списка.

Используется для присвоения переменным значений

по умолчанию:

```
// будет height=100, если переменная height равна null или
undefined
height = height ?? 100;
```

Оператор `??` имеет очень низкий приоритет, лишь немного выше, чем у `?` и `=`, поэтому при использовании его в выражении, скорее всего, потребуются скобки.

Запрещено использовать вместе с `||` или `&&` без явно указанных круглых скобок.

Циклы while и for

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз.

Например, вывести товары из списка один за другим. Или просто перебрать все числа от `1` до `10` и для каждого выполнить одинаковый код.

Для многократного повторения одного участка кода предусмотрены *циклы*.

Цикл «while»

Цикл `while` имеет следующий синтаксис:

```
while (condition) {
  // код
  // также называемый "телом цикла"
}
```

Код из тела цикла выполняется, пока условие `condition`

истинно.

Например, цикл ниже выводит `i`, пока `i < 3`:

```
let i = 0;
while (i < 3) { // выводит 0, затем 1, затем 2
  alert( i );
  i++;
}
```

Одно выполнение тела цикла по-научному называется *итерация*. Цикл в примере выше совершает три итерации.

Если бы строка `i++` отсутствовала в примере выше, то цикл бы повторялся (в теории) вечно. На практике, конечно, браузер не позволит такому случиться, он предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие `while` вычисляется и преобразуется в логическое значение.

Например, `while (i)` – более краткий вариант `while (i != 0)`:

```
let i = 3;

  alert( i );
  i--;
}
while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится
```

Фигурные скобки не требуются для тела цикла из одной строки

Если тело цикла состоит лишь из одной

инструкции, мы можем опустить фигурные скобки `{…}`:

```
let i = 3;
while (i) alert(i--);
```

Цикл «do…while»

Проверку условия можно разместить под телом цикла, используя специальный синтаксис `do..while`:

```
do {
  // тело цикла
} while (condition);
```

Цикл сначала выполнит тело, а затем проверит условие `condition`, и пока его значение равно `true`, он будет выполняться снова и снова.

Например:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось **хотя бы один раз**, даже если условие окажется ложным. На практике чаще используется форма с предусловием: `while(…) {…}`.

Цикл «for»

Более сложный, но при этом самый распространённый цикл — цикл `for`.

Выглядит он так:

```
for (начало; условие; шаг) {
  // ... тело цикла ...
}
```

Давайте разберёмся, что означает каждая часть, на примере. Цикл ниже выполняет `alert(i)` для `i` от `0` до (но не включая) `3`:

```
for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2
  alert(i);
}
```

Рассмотрим конструкцию `for` подробней:

часть		
начало	`i = 0`	Выполняется один раз при входе в цикл
условие	`i < 3`	Проверяется *перед* каждой итерацией цикла. Если оно вычислится в `false`, цикл остановится.
шаг	`i++`	Выполняется *после* тела цикла на каждой итерации *перед* проверкой условия.
тело	`alert(i)`	Выполняется снова и снова,

пока условие вычисляется в `true`.

В целом, алгоритм работы цикла выглядит следующим образом:

```
Выполнить *начало*
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ (Если *условие* == true → Выполнить *тело*, Выполнить *шаг*)
→ ...
```

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.

Если тема циклов для вас нова, может быть полезным вернуться к примеру выше и воспроизвести его работу на листе бумаги, шаг за шагом.

Вот в точности то, что происходит в нашем случае:

```
// for (let i = 0; i < 3; i++) alert(i)

// Выполнить начало
let i = 0;
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// Если условие == true → Выполнить тело, Выполнить шаг
if (i < 3) { alert(i); i++ }
// ...конец, потому что теперь i == 3
```

Встроенное объявление переменной

В примере переменная счётчика `i` была объявлена прямо в цикле. Это так называемое

«встроенное» объявление переменной. Такие переменные существуют только внутри цикла.

```
for ( i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // ошибка, нет такой переменной
let
```

Вместо объявления новой переменной мы можем использовать уже существующую:

```
let i = 0;

for (i = 0; i < 3; i++) { // используем существующую
переменную
  alert(i); // 0, 1, 2
}

alert(i); // 3, переменная доступна, т.к. была
объявлена снаружи цикла
```

Пропуск частей «for»

Любая часть `for` может быть пропущена.

Для примера, мы можем пропустить `начало` если нам ничего не нужно делать перед стартом цикла.

Вот так:

```
let i = 0; // мы уже имеем объявленную i с присвоенным
значением

for (; i < 3; i++) { // нет необходимости в "начале"
  alert( i ); // 0, 1, 2
}
```

Можно убрать и `шаг`:

```
let i = 0;
```

```
for (; i < 3;) {
  alert( i++ );
}
```

Это сделает цикл аналогичным `while (i < 3)`.

А можно и вообще убрать всё, получив бесконечный цикл:

```
for (;;) {
  // будет выполняться вечно
}
```

При этом сами точки с запятой `;` обязательно должны присутствовать, иначе будет ошибка синтаксиса.

Прерывание цикла: «break»

Обычно цикл завершается при вычислении *условия* в `false`.

Но мы можем выйти из цикла в любой момент с помощью специальной директивы `break`.

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

```
let sum = 0;

while (true) {

  let value = +prompt("Введите число", '');



  sum += value;

}
alert( 'Сумма: ' + sum );
  if (!value) break; // (*)
```

Директива `break` в строке `(*)` полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на `alert`.

Вообще, сочетание «бесконечный цикл + `break`» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине.

Переход к следующей итерации: continue

Директива `continue` – «облегчённая версия» `break`. При её выполнении цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно `true`).

Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

Например, цикл ниже использует `continue`, чтобы выводить только нечётные значения:

```
for (let i = 0; i < 10; i++) {

  // если true, пропустить оставшуюся часть тела цикла


  alert(i); // 1, затем 3, 5, 7, 9
}
if
 (i % 2 == 0) continue;
```

Для чётных значений `i`, директива `continue` прекращает выполнение тела цикла и передаёт управление на следующую итерацию `for` (со следующим числом). Таким образом `alert` вызывается только для нечётных значений.

Директива `continue` позволяет избегать вложенности

Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

```
for (let i = 0; i < 10; i++) {

  if (i % 2) {
    alert( i );
  }

}
```

С технической точки зрения он полностью идентичен. Действительно, вместо `continue` можно просто завернуть действия в блок `if`.

Однако мы получили дополнительный уровень вложенности фигурных скобок. Если код внутри `if` более длинный, то это ухудшает читаемость, в отличие от варианта с `continue`.

Нельзя использовать `break/continue` справа от оператора „?“

Обратите внимание, что эти синтаксические конструкции не являются выражениями и не могут быть использованы с тернарным оператором `?`. В частности, использование таких директив, как `break/continue`, вызовет ошибку.

Например, если мы возьмём этот код:

```
if (i > 5) {
  alert(i);
} else {
  continue;
```

```
}
```

…и перепишем его, используя вопросительный знак:

```
(i > 5) ? alert(i) : ; // continue здесь приведёт к ошибке
continue
```

…то будет синтаксическая ошибка.

Это ещё один повод не использовать оператор вопросительного знака `?` вместо `if`.

Метки для break/continue

Бывает, нужно выйти одновременно из нескольких уровней цикла сразу.

Например, в коде ниже мы проходимся циклами по `i` и `j`, запрашивая с помощью `prompt` координаты `(i, j)` с `(0,0)` до `(2,2)`:

```
for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Значение на координатах (${i},${j})`, '');

    // Что если мы захотим перейти к Готово (ниже) прямо отсюда?

  }
}

alert('Готово!');
```

Нам нужен способ остановить выполнение если пользователь отменит ввод.

Обычный `break` после `input` лишь прервёт внутренний цикл, но этого недостаточно. Достичь желаемого поведения можно с помощью меток.

Метка имеет вид идентификатора с двоеточием перед циклом:

```
labelName: for (...) {
  ...
}
```

Вызов `break <labelName>` в цикле ниже ищет ближайший внешний цикл с такой меткой и переходит в его конец.

```
 for (let i = 0; i < 3; i++) {

  for (let j = 0; j < 3; j++) {

    let input = prompt(`Значение на координатах (${i},${j})`, '');

    // если пустая строка или Отмена, то выйти из обоих циклов
    if (!input) ; // (*)

    // сделать что-нибудь со значениями...
  }
}

alert('Готово!');
```

```
outer:
break
 outer
```

В примере выше это означает, что вызовом `break outer` будет разорван внешний цикл до метки с именем `outer`, и управление перейдёт со строки, помеченной `(*)`, к `alert('Готово!')`.

Можно размещать метку на отдельной строке:

```
outer:
```

```
for (let i = 0; i < 3; i++) { ... }
```

Директива `continue` также может быть использована с меткой. В этом случае управление перейдёт на следующую итерацию цикла с меткой.

Метки не позволяют «прыгнуть» куда угодно

Метки не дают возможности передавать управление в произвольное место кода.

Например, нет возможности сделать следующее:

```
break label; // не прыгает к метке ниже

label: for (...)
```

Вызов `break/continue` возможен только внутри цикла, и метка должна находиться где-то выше этой директивы.

Итого

Мы рассмотрели 3 вида циклов:

- `while` – Проверяет условие перед каждой итерацией.
- `do..while` – Проверяет условие после каждой итерации.
- `for (;;)` – Проверяет условие перед каждой итерацией, есть возможность задать дополнительные настройки.

Чтобы организовать бесконечный цикл, используют

конструкцию `while (true)`. При этом он, как и любой другой цикл, может быть прерван директивой `break`.

Если на данной итерации цикла делать больше ничего не надо, но полностью прекращать цикл не следует – используют директиву `continue`.

Обе этих директивы поддерживают *метки*, которые ставятся перед циклом. Метки – единственный способ для `break/continue` выйти за пределы текущего цикла, повлиять на выполнение внешнего.

Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

Задачи

Последнее значение цикла

важность: 3

Какое последнее значение выведет этот код? Почему?

```
let i = 3;

while (i) {
  alert( i-- );
}
```

К решению

Какие значения выведет цикл while?

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

Оба цикла выводят `alert` с одинаковыми значениями или нет?

1. Префиксный вариант `++i`:

```
let i = 0;
while (++i < 5) alert( i );
```

2. Постфиксный вариант `i++`

```
let i = 0;
while (i++ < 5) alert( i );
```

К решению

Какие значения выведет цикл for?

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

Оба цикла выведут `alert` с одинаковыми значениями или нет?

1. Постфиксная форма:

```
for (let i = 0; i < 5; i++) alert( i );
```

2. Префиксная форма:

```
for (let i = 0; i < 5; ++i) alert( i );
```

К решению

Выведите чётные числа

важность: 5

При помощи цикла `for` выведите чётные числа от `2` до `10`.

Запустить демо

К решению

Замените for на while

важность: 5

Перепишите код, заменив цикл `for` на `while`, без изменения поведения цикла.

```
for (let i = 0; i < 3; i++) {
  alert( `number ${i}!` );
}
```

К решению

Повторять цикл, пока ввод неверен

важность: 5

Напишите цикл, который предлагает `prompt` ввести число, большее `100`. Если посетитель ввёл другое число – попросить ввести ещё раз, и так далее.

Цикл должен спрашивать число пока либо посетитель не введёт число, большее `100`, либо не нажмёт кнопку Отмена (ESC).

Предполагается, что посетитель вводит только числа. Предусматривать обработку нечисловых строк в этой задаче необязательно.

Запустить демо

К решению

Вывести простые числа

важность: 3

Натуральное число, большее `1`, называется простым, если оно ни на что не делится, кроме себя и `1`.

Другими словами, `n > 1` – простое, если при его делении на любое число кроме `1` и `n` есть остаток.

Например, `5` это простое число, оно не может быть разделено без остатка на `2`, `3` и `4`.

Напишите код, который выводит все простые числа из интервала от `2` до `n`.

Для `n = 10` результат должен быть `2,3,5,7`.

P.S. Код также должен легко модифицироваться для любых других интервалов.

К решению

Конструкция "switch"

Конструкция `switch` заменяет собой сразу несколько `if`.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Синтаксис

Конструкция `switch` имеет один или более блок `case` и необязательный блок `default`.

Выглядит она так:

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    [break]

  case 'value2':  // if (x === 'value2')
    ...
    [break]

  default:
    ...
    [break]
}
```

Переменная `x` проверяется на строгое равенство первому значению `value1`, затем второму `value2` и так далее.

Если соответствие установлено – `switch` начинает выполняться от соответствующей директивы `case` и далее, до ближайшего `break` (или до конца `switch`).

Если ни один `case` не совпал – выполняется (если есть) вариант `default`.

Пример работы

Пример использования `switch` (сработавший код выделен):

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Маловато' );
    break;

  case 5:
    alert( 'Перебор' );
    break;
  default:
    alert( "Нет таких значений" );
}
```

```
  case 4:
    alert( 'В точку!' );
    break;
```

Здесь оператор `switch` последовательно сравнит `a` со всеми вариантами из `case`.

Сначала `3`, затем – так как нет совпадения – `4`. Совпадение найдено, будет выполнен этот вариант, со строки `alert( 'В точку!' )` и далее, до ближайшего `break`, который прервёт выполнение.

Если `break` нет, то выполнение пойдёт ниже по следующим `case`, при этом остальные проверки игнорируются.

Пример без `break`:

```
let a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Маловато' );

}
```

```
  case 4:
    alert( 'В точку!' );
  case 5:
    alert( 'Перебор' );
  default:
    alert( "Нет таких значений" );
```

В примере выше последовательно выполнятся три `alert`:

```
alert( 'В точку!' );
alert( 'Перебор' );
alert( "Нет таких значений" );
```

Любое выражение может быть аргументом для `switch/case`

И `switch` и `case` допускают любое выражение в качестве аргумента.

Например:

```
let a = "1";
let b = 0;

switch (+a) {


  default:
    alert("Это не выполнится");
}
  case b + 1:
    alert("Выполнится, т.к. значением
+a будет 1, что в точности равно b+1");
    break;
```

В этом примере выражение `+a` вычисляется в `1`, что совпадает с выражением `b + 1` в `case`, и следовательно, код в этом блоке будет

выполнен.

Группировка «case»

Несколько вариантов `case`, использующих один код, можно группировать.

Для примера, выполним один и тот же код для `case 3` и `case 5`, сгруппировав их:

```
let a = 2 + 2;

switch (a) {
  case 4:
    alert('Правильно!');
    break;



  default:
    alert('Результат выглядит странновато. Честно.');
}
   case 3: // (*) группируем оба case
   case 5:
     alert('Неправильно!');
     alert("Может вам посетить урок
математики?");
     break;
```

Теперь оба варианта `3` и `5` выводят одно сообщение.

Возможность группировать `case` – это побочный эффект того, как `switch/case` работает без `break`. Здесь выполнение `case 3` начинается со строки `(*)` и продолжается в `case 5`, потому что отсутствует `break`.

Тип имеет значение

Нужно отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство.

Для примера, давайте рассмотрим следующий код:

```
let arg = prompt("Введите число?");
switch (arg) {
  case '0':
  case '1':
    alert( 'Один или ноль' );
    break;

  case '2':
    alert( 'Два' );
    break;

  case 3:
    alert( 'Никогда не выполнится!' );
    break;
  default:
    alert( 'Неизвестное значение' );
}
```

1. Для `'0'` и `'1'` выполнится первый `alert`.
2. Для `'2'` – второй `alert`.
3. Но для `3`, результат выполнения `prompt` будет строка `"3"`, которая не соответствует строгому равенству `===` с числом `3`. Таким образом, мы имеем «мёртвый код» в `case 3`! Выполнится вариант `default`.

Задачи

Напишите "if", аналогичный "switch"

важность: 5

Напишите `if..else`, соответствующий следующему `switch`:

```
switch (browser) {
  case 'Edge':
    alert( "You've got the Edge!" );
    break;

  case 'Chrome':
  case 'Firefox':
  case 'Safari':
  case 'Opera':
    alert( 'Okay we support these browsers too' );
    break;

  default:
    alert( 'We hope that this page looks ok!' );
}
```

К решению

Переписать условия "if" на "switch"

важность: 4

Перепишите код с использованием одной конструкции `switch`:

```
const number = +prompt('Введите число между 0 и 3', '');

if (number === 0) {
  alert('Вы ввели число 0');
}

if (number === 1) {
  alert('Вы ввели число 1');
}

if (number === 2 || number === 3) {
  alert('Вы ввели число 2, а может и 3');
}
```

К решению

Функции

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, необходимо красиво вывести сообщение при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

Примеры встроенных функций вы уже видели – это `alert(message)`, `prompt(message, default)` и `confirm(question)`. Но можно создавать и свои.

Объявление функции

Для создания функций мы можем использовать *объявление функции*.

Пример объявления функции:

```
function showMessage() {
  alert( 'Всем привет!' );
}
```

Вначале идёт ключевое слово `function`, после него *имя функции*, затем список *параметров* в круглых скобках через запятую (в вышеприведённом примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

```
function имя(параметры) {
  ...тело...
}
```

Наша новая функция может быть вызвана по её имени: `showMessage()`.

Например:

```
function showMessage() {
  alert( 'Всем привет!' );
}

showMessage();
showMessage();
```

Вызов `showMessage()` выполняет код функции. Здесь мы увидим сообщение дважды.

Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции.

Например:

```
function showMessage() {


  alert( message );
}

showMessage(); // Привет, я JavaScript!

alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции
```

```
  let message = "Привет, я JavaScript!"; // локальная переменная
```

Внешние переменные

У функции есть доступ к внешним переменным, например:

```
let  = 'Вася';

function showMessage() {
  let message = 'Привет, ' + ;
  alert(message);
}

showMessage(); // Привет, Вася
```

userName

userName

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Например:

```
let  = 'Вася';

function showMessage() {
   = "Петя"; // (1) изменяем значение внешней переменной

  let message = 'Привет, ' + ;
  alert(message);
}

alert( userName ); //  перед вызовом функции

showMessage();

alert( userName ); // , значение внешней переменной было изменено функцией
```

userName

userName

userName

Вася

```
Петя
```

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную `userName`. Внешняя будет проигнорирована:

```
let userName = 'Вася';

function showMessage() {


  let message = 'Привет, ' + userName; //
  alert(message);
}

// функция создаст и будет использовать свою собственную локальную переменную userName
showMessage();

alert( userName ); // , не изменилась, функция не трогала внешнюю переменную
  let userName = "Петя"; // объявляем локальную переменную
Петя
Вася
```

Глобальные переменные

Переменные, объявленные снаружи всех функций, такие как внешняя переменная `userName` в вышеприведённом коде – называются *глобальными*.

Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных.

Параметры

Мы можем передать внутрь функции любую информацию, используя параметры (также называемые *аргументами функции*).

В нижеприведённом примере функции передаются два параметра: `from` и `text`.

```
function showMessage() { // аргументы: from, text
  alert(from + ': ' + text);
}

from
, text
showMessage('Аня', 'Привет!'); // Аня: Привет!
(*)
showMessage('Аня', "Как дела?"); // Аня: Как
дела? (**)
```

Когда функция вызывается в строках `(*)` и `(**)`, переданные значения копируются в локальные переменные `from` и `text`. Затем они используются в теле функции.

Вот ещё один пример: у нас есть переменная `from`, и мы передаём её функции. Обратите внимание: функция изменяет значение `from`, но это изменение не видно снаружи. Функция всегда получает только копию значения:

```
function showMessage(from, text) {
```

```
  alert( from + ': ' + text );
}

let from = "Аня";

showMessage(from, "Привет"); // *Аня*: Привет

// значение "from" осталось прежним, функция изменила значение локальной переменной
alert( from ); // Аня
  from = '*' + from + '*'; // немного украсим "from"
```

Параметры по умолчанию

Если параметр не указан, то его значением становится `undefined`.

Например, вышеупомянутая функция `showMessage(from, text)` может быть вызвана с одним аргументом:

```
showMessage("Аня");
```

Это не приведёт к ошибке. Такой вызов выведет `"Аня: undefined"`. В вызове не указан параметр `text`, поэтому предполагается, что `text === undefined`.

Если мы хотим задать параметру `text` значение по умолчанию, мы должны указать его после `=`:

```
function showMessage(from, ) {
  alert( from + ": " + text );
}

showMessage("Аня"); // Аня: текст не добавлен
text = "текст не добавлен"
```

Теперь, если параметр `text` не указан, его значением

будет `"текст не добавлен"`

В данном случае `"текст не добавлен"` это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() выполнится только если не передан text
  // результатом будет значение text
}
```

Вычисление параметров по умолчанию

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра.

В примере выше `anotherFunction()` будет вызываться каждый раз, когда `showMessage()` вызывается без параметра `text`.

Использование параметров по умолчанию в ранних версиях JavaScript

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах.

Например, явная проверка на `undefined`:

```
function showMessage(from, text) {

  alert( from + ": " + text );
```

```
  }
    if (text === undefined) {
      text = 'текст не добавлен';
    }
```

...Или с помощью оператора ||:

```
function showMessage(from, text) {
  // Если значение text ложно, тогда присвоить
параметру text значение по умолчанию
  text = text || 'текст не добавлен';
  ...
}
```

Возврат значения

Функция может вернуть результат, который будет передан в вызвавший её код.

Простейшим примером может служить функция сложения двух чисел:

```
function sum(a, b) {
   a + b;
}

let result = sum(1, 2);
alert( result ); // 3
return
```

Директива `return` может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной `result` выше).

Вызовов `return` может быть несколько, например:

```
function checkAge(age) {
```

```
  if (age > 18) {

  } else {

  }
}

let age = prompt('Сколько вам лет?', 18);

if ( checkAge(age) ) {
  alert( 'Доступ получен' );
} else {
  alert( 'Доступ закрыт' );
}
```

```
    return true;
    return confirm('А родители разрешили?');
```

Возможно использовать `return` и без значения. Это приведёт к немедленному выходу из функции.

Например:

```
function showMovie(age) {
  if ( !checkAge(age) ) {

  }

  alert( "Вам показывается кино" ); // (*)
  // ...
}
```

```
    return;
```

В коде выше, если `checkAge(age)` вернёт `false`, `showMovie` не выполнит `alert`.

Результат функции с пустым `return` или без него – `undefined`

Если функция не возвращает значения, это всё равно, как если бы она возвращала `undefined`:

```
function doNothing() { /* пусто */ }

alert( doNothing() === undefined ); // true
```

Пустой `return` аналогичен `return undefined`:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```

Никогда не добавляйте перевод строки между `return` и его значением

Для длинного выражения в `return` может быть заманчиво разместить его на нескольких отдельных строках, например так:

```
return
 (some + long + expression + or + whatever * f(a) +
f(b))
```

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после `return`. Для него это будет выглядеть так:

```
return
 (some + long + expression + or + whatever * f(a) +
f(b))
;
```

Таким образом, это фактически стало пустым `return`.

Если мы хотим, чтобы возвращаемое выражение

занимало несколько строк, нужно начать его на той же строке, что и `return`. Или, хотя бы, поставить там открывающую скобку, вот так:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

И тогда всё сработает, как задумано.

Выбор имени функции

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции, чтобы программист, который будет читать код, получил верное представление о том, что делает функция.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов.

Например, функции, начинающиеся с `"show"` обычно что-то показывают.

Функции, начинающиеся с…

- `"get…"` – возвращают значение,
- `"calc…"` – что-то вычисляют,
- `"create…"` – что-то создают,
- `"check…"` – что-то проверяют и возвращают логическое значение, и т.д.

Примеры таких имён:

```
showMessage(..)      // показывает сообщение
getAge(..)           // возвращает возраст (в каком либо
значении)
calcSum(..)          // вычисляет сумму и возвращает результат
createForm(..)       // создаёт форму (и обычно возвращает её)
checkPermission(..) // проверяет доступ, возвращая true/false
```

Благодаря префиксам, при первом взгляде на имя функции становится понятным что делает её код, и какое значение она может возвращать.

Одна функция – одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием.

Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Несколько примеров, которые нарушают это правило:

- `getAge` – будет плохим выбором, если функция будет выводить `alert` с возрастом (должна только возвращать его).
- `createForm` – будет плохим выбором, если функция будет изменять документ, добавляя форму в него (должна только создавать форму и возвращать её).
- `checkPermission` – будет плохим выбором, если функция будет отображать сообщение с текстом `доступ разрешён/запрещён`

(должна только выполнять проверку и возвращать её результат).

В этих примерах использовались общепринятые смыслы префиксов. Конечно, вы в команде можете договориться о других значениях, но обычно они мало отличаются от общепринятых. В любом случае вы и ваша команда должны точно понимать, что значит префикс, что функция с ним может делать, а чего не может.

Сверхкороткие имена функций

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Например, во фреймворке jQuery есть функция с именем `$`. В библиотеке Lodash основная функция представлена именем `_`.

Это исключения. В основном имена функций должны быть в меру краткими и описательными.

Функции == Комментарии

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Иногда следовать этому правилу непросто, но это определённо хорошее правило.

Небольшие функции не только облегчают тестирование и отладку – само существование таких функций выполняет роль хороших комментариев!

Например, сравним ниже две функции `showPrimes(n)`.

Каждая из них выводит простое число до `n`.

Первый вариант использует метку `nextPrime`:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // простое
  }
}
```

Второй вариант использует дополнительную функцию `isPrime(n)` для проверки на простое:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {


    alert(i);  // простое
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
if
  (!isPrime(i)) continue;
```

Второй вариант легче для понимания, не правда ли? Вместо куска кода мы видим название действия (`isPrime`). Иногда разработчики называют такой код *самодокументируемым*.

Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

Итого

Объявление функции имеет вид:

```
function имя(параметры, через, запятую) {
  /* тело, код функции */
}
```

- Передаваемые значения копируются в параметры функции и становятся локальными переменными.
- Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.
- Функция может возвращать значение. Если этого не происходит, тогда результат равен `undefined`.

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.

Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.

Именование функций:

- Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
- Функция – это действие, поэтому её имя обычно является глаголом.
- Есть много общепринятых префиксов, таких как: `create…`, `show…`, `get…`, `check…` и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.

Функции являются основными строительными блоками скриптов. Мы рассмотрели лишь основы функций в JavaScript, но уже сейчас можем создавать и использовать их. Это только начало пути. Мы будем неоднократно возвращаться к функциям и изучать их всё более и более глубоко.

Задачи

Обязателен ли "else"?

важность: 4

Следующая функция возвращает `true`, если параметр `age` больше `18`.

В ином случае она запрашивает подтверждение через `confirm` и возвращает его результат:

```
function checkAge(age) {
  if (age > 18) {
    return true;

}
   } else {
     // ...
     return confirm('Родители разрешили?');
   }
```

Будет ли эта функция работать как-то иначе, если убрать `else`?

```
function checkAge(age) {
  if (age > 18) {
    return true;
  }

}
```

```
  // ...
  return confirm('Родители разрешили?');
```

Есть ли хоть одно отличие в поведении этого варианта?

К решению

Перепишите функцию, используя оператор '?' или '||'

важность: 4

Следующая функция возвращает `true`, если параметр `age` больше `18`.

В ином случае она задаёт вопрос `confirm` и возвращает его результат.

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Родители разрешили?');
  }
}
```

Перепишите функцию, чтобы она делала то же самое, но без `if`, в одну строку.

Сделайте два варианта функции `checkAge`:

1. Используя оператор `?`
2. Используя оператор `||`

К решению

Функция min(a, b)

важность: 1

Напишите функцию `min(a,b)`, которая возвращает меньшее из чисел `a` и `b`.

Пример вызовов:

```
min(2, 5) == 2
min(3, -1) == -1
min(1, 1) == 1
```

К решению

Функция pow(x,n)

важность: 4

Напишите функцию `pow(x,n)`, которая возвращает `x` в степени `n`. Иначе говоря, умножает `x` на себя `n` раз и возвращает результат.

```
pow(3, 2) = 3 * 3 = 9
pow(3, 3) = 3 * 3 * 3 = 27
pow(1, 100) = 1 * 1 * ...* 1 = 1
```

Создайте страницу, которая запрашивает `x` и `n`, а затем выводит результат `pow(x,n)`.

Запустить демо

P.S. В этой задаче функция обязана поддерживать только натуральные значения `n`, т.е. целые от `1` и выше.

К решению

Function Expression

Функция в JavaScript – это не магическая языковая структура, а особого типа значение.

Синтаксис, который мы использовали до этого, называется *Function Declaration* (Объявление Функции):

```
function sayHi() {
  alert( "Привет" );
}
```

Существует ещё один синтаксис создания функций, который называется *Function Expression* (Функциональное Выражение).

Оно выглядит вот так:

```
let sayHi = function() {
  alert( "Привет" );
};
```

В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. По сути без разницы, как мы определили функцию, это просто значение, хранимое в переменной `sayHi`.

Смысл обоих примеров кода одинаков: "создать функцию и поместить её значение в переменную `sayHi`".

Мы можем даже вывести это значение с помощью `alert`:

```
function sayHi() {
  alert( "Привет" );
}
```

```
alert( sayHi ); // выведет код функции
```

Обратите внимание, что последняя строка не вызывает функцию `sayHi`, после её имени нет круглых скобок. Существуют языки программирования, в которых любое упоминание имени функции совершает её вызов. JavaScript – не один из них.

В JavaScript функции – это значения, поэтому мы и обращаемся с ними, как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.

Конечно, функция – не обычное значение, в том смысле, что мы можем вызвать его при помощи скобок: `sayHi()`.

Но всё же это значение. Поэтому мы можем делать с ним то же самое, что и с любым другим значением.

Мы можем скопировать функцию в другую переменную:

```
function sayHi() {   // (1) создаём
  alert( "Привет" );
}

let func = sayHi;    // (2) копируем

func(); // Привет    // (3) вызываем копию (работает)!
sayHi(); // Привет   //     прежняя тоже работает (почему бы нет)
```

Давайте подробно разберём всё, что тут произошло:

1. Объявление Function Declaration `(1)` создало функцию и присвоило её значение переменной с именем `sayHi`.
2. В строке `(2)` мы скопировали её значение в переменную `func`. Обратите внимание (ещё раз): нет круглых скобок после `sayHi`. Если бы они были, то выражение `func = sayHi()` записало бы *результат вызова* `sayHi()` в переменную `func`, а не саму

функцию `sayHi`.

3. Теперь функция может быть вызвана с помощью обеих переменных `sayHi()` и `func()`.

Заметим, что мы могли бы использовать и Function Expression для того, чтобы создать `sayHi` в первой строке:

```
let sayHi = function() {
  alert( "Привет" );
};

let func = sayHi;
// ...
```

Результат был бы таким же.

Зачем нужна точка с запятой в конце?

У вас мог возникнуть вопрос: Почему в Function Expression ставится точка с запятой `;` на конце, а в Function Declaration нет:

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
}
;
```

Ответ прост:

> Нет необходимости в `;` в конце блоков кода и синтаксических конструкций, которые их используют, таких как `if { ... }`, `for { }`, `function f { }` и т.д.

Function Expression использует внутри себя инструкции присваивания `let sayHi = ...;` как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.

Функции-«колбэки»

Рассмотрим ещё примеры функциональных выражений и передачи функции как значения.

Давайте напишем функцию `ask(question, yes, no)` с тремя параметрами:

`question`

Текст вопроса

`yes`

Функция, которая будет вызываться, если ответ будет «Yes»

`no`

Функция, которая будет вызываться, если ответ будет «No»

Наша функция должна задать вопрос `question` и, в зависимости от того, как ответит пользователь, вызвать `yes()` или `no()`:

```
function showOk() {
  alert( "Вы согласны." );
}

function showCancel() {
  alert( "Вы отменили выполнение." );
}

// использование: функции showOk, showCancel передаются в
качестве аргументов ask
ask("Вы согласны?", showOk, showCancel);
function ask(question, yes, no) {
```

```
  if (confirm(question)) yes()
  else no();
}
```

На практике подобные функции очень полезны. Основное отличие «реальной» функции `ask` от примера выше будет в том, что она использует более сложные способы взаимодействия с пользователем, чем простой вызов `confirm`. В браузерах такие функции обычно отображают красивые диалоговые окна. Но это уже другая история.

Аргументы функции `ask` ещё называют *функциями-колбэками* или просто *колбэками*.

Ключевая идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. В нашем случае, `showOk` становится *колбэком*' для ответа «yes», а `showCancel` – для ответа «no».

Мы можем переписать этот пример значительно короче, используя Function Expression:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили
выполнение."); }
);
```

Здесь функции объявляются прямо внутри вызова `ask(...)`. У них нет имён, поэтому они называются *анонимными*. Такие функции недоступны снаружи `ask` (потому что они не присвоены переменным), но это как раз

то, что нам нужно.

Подобный код, появившийся в нашем скрипте выглядит очень естественно, в духе JavaScript.

> **Функция – это значение, представляющее «действие»**
>
> Обычные значения, такие как строки или числа представляют собой *данные*.
>
> Функции, с другой стороны, можно воспринимать как «действия».
>
> Мы можем передавать их из переменной в переменную и запускать, когда захотим.

Function Expression в сравнении с Function Declaration

Давайте разберём ключевые отличия Function Declaration от Function Expression.

Во-первых, синтаксис: как определить, что есть что в коде.

Function Declaration: функция объявляется отдельной конструкцией «function…» в основном потоке кода.

```
// Function Declaration
function sum(a, b) {
  return a + b;
}
```

Function Expression: функция, созданная внутри другого выражения или синтаксической конструкции. В данном случае функция создаётся в правой части «выражения

присваивания» =:

```
// Function Expression
let sum = function(a, b) {
  return a + b;
};
```

Более тонкое отличие состоит, в том, *когда* создаётся функция движком JavaScript.

Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться.

После того, как поток выполнения достигнет правой части выражения присваивания `let sum = function…` – с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д.).

С Function Declaration всё иначе.

Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке).

Другими словами, когда движок JavaScript *готовится* выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Можно считать этот процесс «стадией инициализации».

И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение.

В результате, функции, созданные, как Function Declaration могут быть вызваны раньше своих определений.

Например, так будет работать:

```
function sayHi(name) {
```

```
  alert( `Привет, ${name}` );
}
sayHi("Вася"); // Привет, Вася
```

Функция `sayHi` была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте.

…Если бы это было Function Expression, то такой код вызовет ошибку:

```
let sayHi = function(name) {  // (*) магии больше нет
  alert( `Привет, ${name}` );
};
sayHi("Вася"); // ошибка!
```

Функции, объявленные при помощи Function Expression, создаются тогда, когда выполнение доходит до них. Это случится только на строке, помеченной звёздочкой `(*)`. Слишком поздно.

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости.

В строгом режиме, когда Function Declaration находится в блоке `{...}`, функция доступна везде внутри блока. Но не снаружи него.

Для примера давайте представим, что нам нужно создать функцию `welcome()` в зависимости от значения переменной `age`, которое мы получим во время выполнения кода. И затем запланируем использовать её когда-нибудь в будущем.

Такой код, использующий Function Declaration, работать не будет:

```
let age = prompt("Сколько Вам лет?", 18);
```

```
// в зависимости от условия объявляем функцию
if (age < 18) {

  function welcome() {
    alert("Привет!");
  }

} else {

  function welcome() {
    alert("Здравствуйте!");
  }

}

// ...не работает
welcome(); // Error: welcome is not defined
```

Это произошло, так как объявление Function Declaration видимо только внутри блока кода, в котором располагается.

Вот ещё один пример:

```
let age = 16; // присвоим для примера 16

if (age < 18) {

                            //  |
  function welcome() {      //  |
    alert("Привет!");       //  |  Function Declaration
доступно
  }                         //  |  во всём блоке кода, в
котором объявлено
                            //  |


} else {

  function welcome() {
    alert("Здравствуйте!");
  }
}

// здесь фигурная скобка закрывается,
// поэтому Function Declaration, созданные внутри блока кода
выше -- недоступны отсюда.
```

```
  welcome();                  // \    (выполнится)
  welcome();                  // /    (выполнится)
welcome(); // Ошибка: welcome is not defined
```

Что можно сделать, чтобы `welcome` была видима снаружи `if`?

Верным подходом будет воспользоваться функцией, объявленной при помощи Function Expression, и присвоить значение `welcome` переменной, объявленной снаружи `if`, что обеспечит нам нужную видимость.

Такой код работает, как ожидалось:

```
let age = prompt("Сколько Вам лет?", 18);

let welcome;

if (age < 18) {

  welcome = function() {
    alert("Привет!");
  };

} else {

  welcome = function() {
    alert("Здравствуйте!");
  };

}

welcome(); // теперь всё в порядке
```

Можно упростить этот код ещё сильнее, используя условный оператор `?`:

```
let age = prompt("Сколько Вам лет?", 18);

let welcome = (age < 18) ?
  function() { alert("Привет!"); } :
  function() { alert("Здравствуйте!"); };

welcome(); // теперь всё в порядке
```

Когда использовать Function Declaration, а когда Function Expression?

Как правило, если нам понадобилась функция, в первую очередь нужно рассматривать синтаксис Function Declaration, который мы использовали до этого. Он даёт нам больше свободы в том, как мы можем организовывать код. Функции, объявленные таким образом, можно вызывать до их объявления.

Также функции вида `function f(…) {…}` чуть более заметны в коде, чем `let f = function(…) {…}`. Function Declaration легче «ловятся глазами».

…Но если Function Declaration нам не подходит по какой-то причине (мы рассмотрели это в примере выше), то можно использовать объявление при помощи Function Expression.

Итого

Функции – это значения. Они могут быть присвоены, скопированы или объявлены в другом месте кода.

Если функция объявлена как отдельная инструкция в основном потоке кода, то это Function Declaration.

Если функция была создана как часть выражения, то считается, что эта функция объявлена при помощи Function Expression.

Function Declaration обрабатываются перед выполнением блока кода. Они видны во всём блоке.

Функции, объявленные при помощи Function Expression, создаются, только когда поток выполнения достигает их.

В большинстве случаев, когда нам нужно создать функцию, предпочтительно использовать Function Declaration, т.к. функция будет видима до своего объявления в коде. Это позволяет более гибко организовывать код, и улучшает его читаемость.

Таким образом, мы должны прибегать к объявлению функций при помощи Function Expression в случае, когда синтаксис Function Declaration не подходит для нашей задачи. Мы рассмотрели несколько таких примеров в этой главе, и рассмотрим их ещё больше в будущем.

Функции-стрелки, основы

Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.

Он называется «функции-стрелки» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

```
let func = (arg1, arg2, ...argN) => expression
```

…Такой код создаёт функцию `func` с аргументами `arg1..argN` и вычисляет `expression` с правой стороны с их использованием, возвращая результат.

Другими словами, это более короткий вариант такой записи:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Давайте взглянем на конкретный пример:

```
let sum = (a, b) => a + b;

/* Более короткая форма для:

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

То есть, `(a, b) => a + b` задаёт функцию с двумя аргументами `a` и `b`, которая при запуске вычисляет выражение справа `a + b` и возвращает его результат.

Если у нас только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

```
// тоже что и
// let double = function(n) { return n * 2 }


alert( double(3) ); // 6
let double = n => n * 2;
```

Если нет аргументов, указываются пустые круглые скобки:

```
let sayHi = () => alert("Hello!");

sayHi();
```

Функции-стрелки могут быть использованы так же, как и Function Expression.

Например, для динамического создания функции:

```
let age = prompt("Сколько Вам лет?", 18);

let welcome = (age < 18) ?
  () => alert('Привет') :
```

```
  () => alert("Здравствуйте!");

welcome(); // теперь всё в порядке
```

Поначалу функции-стрелки могут показаться необычными и трудночитаемыми, но это быстро пройдёт, как только глаза привыкнут к этим конструкциям.

Они очень удобны для простых однострочных действий, когда лень писать много букв.

Многострочные стрелочные функции

В примерах выше аргументы использовались слева от `=>`, а справа вычислялось выражение с их значениями.

Порой нам нужно что-то посложнее, например, выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать `return` внутри них, как в обычной функции.

Например:

```
let sum = (a, b) => {  // фигурная скобка, открывающая тело многострочной функции
  let result = a + b;

};

alert( sum(1, 2) ); // 3
  return result; // при фигурных скобках для возврата значения нужно явно вызвать return
```

Дальше будет ещё информация

Здесь мы рассмотрели функции-стрелки как способ писать меньше букв. Но это далеко не

всё!

Стрелочные функции обладают другими интересными особенностями. Их изучение требует знания некоторых других возможностей языка JavaScript, поэтому мы вернёмся к стрелочным функциям позже, в главе Повторяем стрелочные функции.

А пока мы можем использовать их для простых однострочных действий и колбэков.

Итого

Функции-стрелки очень удобны для однострочных действий. Они бывают двух типов:

1. Без фигурных скобок: `(...args) => expression` – правая сторона выражение: функция выполняет его и возвращает результат.
2. С фигурными скобками: `(...args) => { body }` – скобки позволяют нам писать многострочные инструкции внутри функции, но при этом необходимо указывать директиву `return`, чтобы вернуть какое-либо значение.

Задачи

Перепишите с использованием функции-стрелки

Замените код Function Expression стрелочной функцией:

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
```

```
  else no();
}

ask(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили выполнение."); }
);
```

К решению

Особенности JavaScript

Давайте кратко повторим изученный материал и отметим наиболее «тонкие» моменты.

Структура кода

Инструкции разделяются точкой с запятой:

```
alert('Привет'); alert('Мир');
```

Как правило, перевод строки также интерпретируется как разделитель, так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

Это так называемая «автоматическая вставка точки с запятой». Впрочем, она не всегда срабатывает, например:

```
alert("После этого сообщения ждите ошибку")

[1, 2].forEach(alert)
```

Большинство руководств по стилю кода рекомендуют ставить точку с запятой после каждой инструкции.

Точка с запятой не требуется после блоков кода {…} и синтаксических конструкций с ними, таких как, например, циклы:

```
function f() {
  // после объявления функции необязательно ставить точку с запятой
}

for(;;) {
  // после цикла точка с запятой также необязательна
}
```

…Впрочем, если даже мы и поставим «лишнюю» точку с запятой, ошибки не будет. Она просто будет проигнорирована.

Подробности: Структура кода.

Строгий режим

Чтобы по максимуму использовать возможности современного JavaScript, все скрипты рекомендуется начинать с добавления директивы `"use strict"`.

```
'use strict';

...
```

Эту директиву следует размещать в первой строке скрипта или в начале тела функции.

Без `"use strict"` код также запустится, но некоторые возможности будут работать в «режиме совместимости» со старыми версиями языка JavaScript. Нам же предпочтительнее современное поведение.

Некоторые конструкции языка (например, классы, которые нам ещё предстоит изучить) включают строгий режим по умолчанию.

Подробности: Строгий режим — "use strict".

Переменные

Можно объявить при помощи:

- `let`
- `const` (константа, т.е. изменению не подлежит)
- `var` (устаревший способ, подробности позже)

Имя переменной может включать:

- Буквы и цифры, однако цифра не может быть первым символом.
- Символы `$` и `_` используются наряду с буквами.
- Иероглифы и символы нелатинского алфавита также допустимы, но обычно не используются.

Переменные типизируются динамически. В них могут храниться любые значения:

```
let x = 5;
x = "Вася";
```

Всего существует 8 типов данных:

- `number` для целых и вещественных чисел,
- `bigint` для работы с целыми числами произвольной длины,
- `string` для строк,
- `boolean` для логических значений истинности или

ложности: `true/false`,

`null` – тип с единственным значением `null`, т.е. «пустое значение» или «значение не существует»,

`undefined` – тип с единственным значением `undefined`, т.е. «значение не задано»,

`object` и `symbol` – сложные структуры данных и уникальные идентификаторы; их мы ещё не изучили.

Оператор `typeof` возвращает тип значения переменной, с двумя исключениями:

```
typeof null == "object" // ошибка в языке
typeof function(){} == "function" // именно для функций
```

Подробности: Переменные, Типы данных.

Взаимодействие с посетителем

В качестве рабочей среды мы используем браузер, так что простейшими функциями взаимодействия с посетителем являются:

`prompt(question, [default])`

Задаёт вопрос `question` и возвращает то, что ввёл посетитель, либо `null`, если посетитель нажал на кнопку «Отмена».

`confirm(question)`

Задаёт вопрос `question` и предлагает выбрать «ОК» или «Отмена». Выбор возвращается в формате `true/false`.

`alert(message)`

Выводит сообщение `message`.

Все эти функции показывают *модальные окна*, они останавливают выполнение кода и не позволяют посетителю взаимодействовать со страницей, пока не будет дан ответ на вопрос.

Например:

```
let userName = prompt("Введите имя", "Алиса");
let isTeaWanted = confirm("Вы хотите чаю?");

alert( "Посетитель: " + userName ); // Алиса
alert( "Чай: " + isTeaWanted ); // true
```

Подробности: Взаимодействие: alert, prompt, confirm.

Операторы

JavaScript поддерживает следующие операторы:

Арифметические

Простые `* + - /`, а также деление по модулю `%` и возведение в степень `**`.

Бинарный плюс `+` объединяет строки. А если одним из операндов является строка, то второй тоже будет конвертирован в строку:

```
alert( '1' + 2 ); // '12', строка
alert( 1 + '2' ); // '12', строка
```

Операторы присваивания

Простые `a = b` и составные `a *= 2`.

Битовые операции

Битовые операторы работают с 32-битными целыми числами на самом низком, побитовом уровне. Подробнее

об их использовании можно прочитать на ресурсе MDN и в разделе Побитовые операторы.

Условный оператор

Единственный оператор с тремя параметрами: `cond ? resultA : resultB`. Если условие `cond` истинно, возвращается `resultA`, иначе – `resultB`.

Логические операторы

Логические И `&&`, ИЛИ `||` используют так называемое «ленивое вычисление» и возвращают значение, на котором оно остановилось (не обязательно `true` или `false`). Логическое НЕ `!` конвертирует операнд в логический тип и возвращает инвертированное значение.

Сравнение

Проверка на равенство `==` значений разных типов конвертирует их в число (за исключением `null` и `undefined`, которые могут равняться только друг другу), так что примеры ниже равны:

```
alert( 0 == false ); // true
alert( 0 == '' ); // true
```

Другие операторы сравнения тоже конвертируют значения разных типов в числовой тип.

Оператор строгого равенства `===` не выполняет конвертирования: разные типы для него всегда означают разные значения.

Значения `null` и `undefined` особенные: они равны `==` только друг другу, но не равны ничему ещё.

Операторы сравнения больше/меньше сравнивают строки посимвольно, остальные типы конвертируются в число.

Другие операторы

Существуют и другие операторы, такие как запятая.

Подробности: Базовые операторы, математика, Операторы сравнения, Логические операторы.

Циклы

Мы изучили три вида циклов:

```
// 1
while (condition) {
  ...
}

// 2
do {
  ...
} while (condition);

// 3
for(let i = 0; i < 10; i++) {
  ...
}
```

Переменная, объявленная в цикле `for(let...)`, видна только внутри цикла. Но мы также можем опустить `let` и переиспользовать существующую переменную.

Директивы `break/continue` позволяют выйти из цикла/текущей итерации. Используйте метки для выхода из вложенных циклов.

Подробности: Циклы while и for.

Позже мы изучим ещё виды циклов для работы с объектами.

Конструкция «switch»

Конструкция «switch» может заменить несколько проверок `if`. При сравнении она использует оператор строгого равенства `===`.

Например:

```
let age = prompt('Сколько вам лет?', 18);

switch (age) {
  case 18:
    alert("Так не сработает"); // результатом prompt является строка, а не число

  case "18":
    alert("А так сработает!");
    break;

  default:
    alert("Любое значение, неравное значению выше");
}
```

Подробности: Конструкция "switch".

Функции

Мы рассмотрели три способа создания функции в JavaScript:

1. Function Declaration: функция в основном потоке кода

```
function sum(a, b) {
  let result = a + b;

  return result;
}
```

2. Function Expression: функция как часть выражения

```
let sum = function(a, b) {
  let result = a + b;

  return result;
};
```

3. Стрелочные функции:

```
// выражение в правой части
let sum = (a, b) => a + b;

// многострочный код в фигурных скобках { ... }, здесь
нужен return:
let sum = (a, b) => {
  // ...
  return a + b;
}

// без аргументов
let sayHi = () => alert("Привет");

// с одним аргументом
let double = n => n * 2;
```

У функций могут быть локальные переменные: т.е. объявленные в теле функции. Такие переменные видимы только внутри функции.

У параметров могут быть значения по умолчанию: `function sum(a = 1, b = 2) {...}`.

Функции всегда что-нибудь возвращают. Если нет оператора `return`, результатом будет `undefined`.

Подробности: Функции, Функции-стрелки, основы.

Далее мы изучим больше

Это был краткий список возможностей JavaScript. На данный момент мы изучили только основы. Далее в учебнике вы найдёте больше особенностей и продвинутых возможностей JavaScript.

Открыть готовый пример в песочнице.

Качество кода

В этой главе объясняются подходы к написанию кода, которые мы будем использовать в дальнейшем при разработке.

Отладка в браузере Chrome

Давайте отвлечёмся от написания кода и поговорим о его отладке.

Отладка – это процесс поиска и исправления ошибок в скрипте. Все современные браузеры и большинство других сред разработки поддерживают инструменты для отладки – специальный графический интерфейс, который сильно упрощает отладку. Он также позволяет по шагам отследить, что именно происходит в нашем коде.

Мы будем использовать браузер Chrome, так как у него достаточно возможностей, в большинстве других браузеров процесс будет схожим.

Панель «Исходный код» («Sources»)

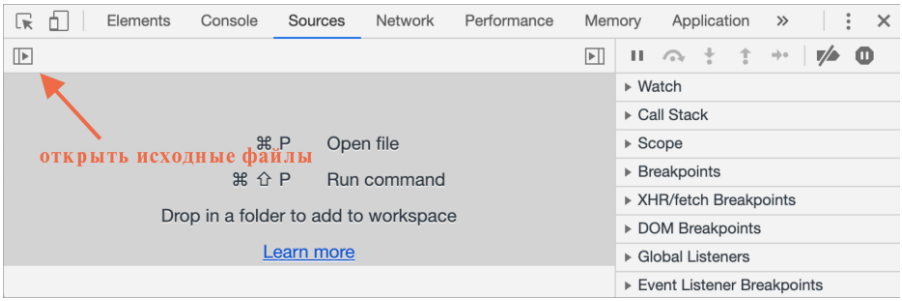
Версия Chrome, установленная у вас, может выглядеть немного иначе, однако принципиальных отличий не будет.

Работая в Chrome, откройте тестовую страницу.

Включите инструменты разработчика, нажав `F12` (Mac: `Cmd+Opt+I`).

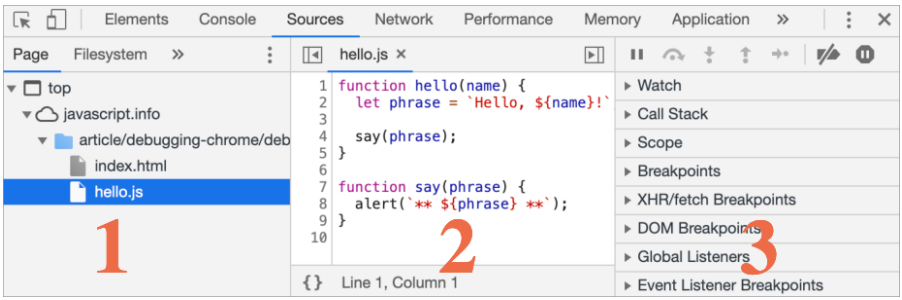
Щёлкните по панели `sources` («исходный код»).

При первом запуске получаем следующее:



Кнопка-переключатель
откроет вкладку со списком файлов.

Кликните на неё и выберите `hello.js`. Вот что появится:



Интерфейс состоит из трёх зон:

1. В зоне **Resources** (Ресурсы) показаны файлы HTML, JavaScript, CSS, включая изображения, используемые на странице. Здесь также могут быть файлы различных расширений Chrome.
2. Зона **Source** показывает исходный код.
3. Наконец, зона **Information and control** (Сведения и контроль) отведена для отладки, вскоре мы к ней вернёмся.

Чтобы скрыть список ресурсов и освободить экранное место для исходного кода, щёлкните по тому же переключателю
.

Консоль

При нажатии на клавишу `Esc` в нижней части экрана вызывается консоль, где можно вводить команды и выполнять их клавишей `Enter`.

Результат выполнения инструкций сразу же отображается в консоли.

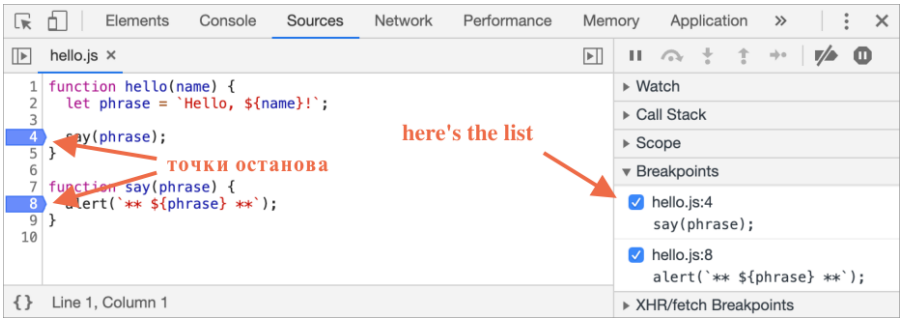
Например, результатом `1+2` будет `3`, а инструкция `hello("debugger")` ничего не возвращает, так что получаем `undefined`:

Точки останова (breakpoints)

Давайте разберёмся, как работает код нашей тестовой страницы. В файле `hello.js` щёлкните по строчке номер `4`. Да, щёлкайте именно по самой цифре, не по коду.

Ура! Вы поставили точку останова. А теперь щёлкните по цифре `8` на восьмой линии. Номер строки будет окрашен в синий цвет.

Вот что в итоге должно получиться:



Точка останова – это участок кода, где отладчик автоматически приостановит исполнение JavaScript.

Пока исполнение поставлено «на паузу», мы можем просмотреть текущие значения переменных, выполнить команды в консоли, одним словом, мы выполняем отладку кода.

В правой части графического интерфейса мы видим список точек останова. А когда таких точек выставлено много, да ещё и в разных файлах, этот список поможет эффективно ими управлять:

Быстро переместиться к любой точке останова в коде – нужно щёлкнуть по точке в правой части экрана.

Временно деактивировать точку – в общем списке снимите галочку напротив ненужной в данный момент точки.

Удалить точку – щёлкните по ней правой кнопкой мыши и выберите Remove (Удалить).

…и так далее.

Условные точки останова

Можно задать и так называемую *условную* точку останова – щёлкните правой кнопкой мыши по номеру строки в коде. Если задать выражение, то именно при его истинности выполнение кода

будет приостановлено.

Этот метод используется, когда выполнение кода нужно остановить при присвоении определённого выражения какой-либо переменной или при определённых параметрах функции.

Команда Debugger

Выполнение кода можно также приостановить с помощью команды `debugger` прямо изнутри самого кода:

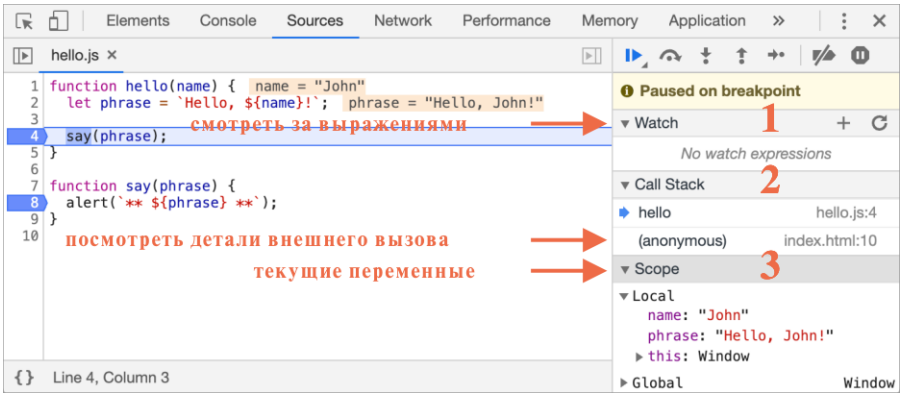
```
function hello(name) {
  let phrase = `Привет, ${name}!`;



  say(phrase);
}
  debugger;  // <-- здесь выполнение
прерывается
```

Способ удобен тем, что можно продолжить работать в редакторе кода без необходимости переключения в браузер для выставления точки останова.

Остановимся и оглядимся

В нашем примере функция `hello()` вызывается во время загрузки страницы, поэтому для начала отладки (после того, как мы поставили точки останова) проще всего её перезагрузить. Нажмите `F5` (Windows, Linux) или `Cmd+R` (Mac).

Выполнение прервётся на четвёртой строчке:



Чтобы понять, что происходит в коде, щёлкните по стрелочкам справа:

1. **`Watch` показывает текущие значения выражений.**

   Нажмите на `+` и введите выражение. В процессе выполнения отладчик автоматически пересчитывает и выводит его значение.

2. **`Call Stack` показывает последовательность вызовов функций.**

   В нашем примере отладчик работает с функцией `hello()`, вызванной скриптом из файла `index.html` (там нет функции, поэтому вызов «анонимный»).

   При нажатии на элемент списка (например, на «anonymous») отладчик переходит к соответствующему коду, и нам представляется возможность его проанализировать.

3. **`Scope` показывает текущие переменные.**

   В `Local` отображаются локальные переменные функций, а их значения подсвечены в исходном коде.

   В `Global` перечисляются глобальные переменные (т.е. объявленные за пределами функций).

   Не обращайте пока внимание на ключевое слово `this`

– его мы изучим чуть позже.

Пошаговое выполнение скрипта

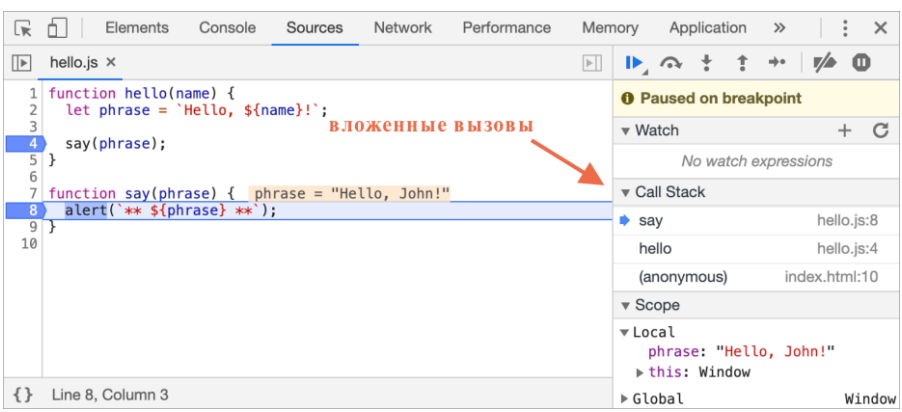
А теперь давайте *пошагаем* по нашему коду.

В правой части панели для этого есть несколько кнопок. Рассмотрим их.

– продолжить выполнение. Быстрая клавиша – `F8`.

Возобновляет выполнение кода. Если больше нет точек останова, отладчик прекращает работу и позволяет приложению работать дальше.

Вот, что мы увидим, кликнув на неё:



Выполнение кода возобновилось, дошло до другой точки останова внутри `say()`, и отладчик снова приостановил выполнение. Обратите внимание на пункт «Call stack» справа: в списке появился ещё один вызов. Мы теперь внутри функции `say()`.

– сделать шаг (выполнить следующую команду), *не заходя в функцию*. Быстрая клавиша – `F10`.

Если мы нажмём на неё – будет вызван `alert`. Важно: на

месте `alert` может быть любая другая функция, выполнение просто *перешагнёт через неё*, полностью игнорируя её содержимое.

– сделать шаг. Быстрая клавиша – `F11`.

В отличие от предыдущего примера, здесь мы «заходим» во вложенные функции и шаг за шагом проходим по скрипту.

– продолжить выполнение до завершения текущей функции. Быстрая клавиша – `Shift+F11`.

Выполнение кода остановится на самой последней строчке текущей функции. Этот метод применяется, когда мы случайно нажали
и зашли в функцию, но нам она неинтересна и мы как можно скорее хотим из неё выбраться.

– активировать/деактивировать все точки останова.

Эта кнопка не влияет на выполнение кода, она лишь позволяет массово включить/отключить точки останова.

– разрешить/запретить остановку выполнения в случае возникновения ошибки.

Если опция включена и инструменты разработчика открыты, любая ошибка в скрипте приостанавливает выполнение кода, что позволяет его проанализировать. Поэтому если скрипт завершается с ошибкой, открываем отладчик, включаем эту опцию, перезагружаем страницу и локализуем проблему.

Continue to here

Если щёлкнуть правой кнопкой мыши по строчке

кода, в контекстном меню можно выбрать опцию «Continue to here» («продолжить до этого места»).

Этот метод используется, когда нам нужно продвинуться на несколько шагов вперёд до нужной строки, но лень выставлять точки останова.

Логирование

Если нужно что-то вывести в консоль из кода, применяется функция `console.log`.

К примеру, выведем в консоль значения от нуля до четырёх:

```
// чтобы увидеть результат, сначала откройте консоль
for (let i = 0; i < 5; i++) {
  console.log("value,", i);
}
```

Обычный пользователь сайта не увидит такой вывод, так как он в консоли. Напомним, что консоль можно открыть через инструменты разработчика – выберите вкладку «Консоль» или нажмите `Esc`, находясь в другой вкладке – консоль откроется в нижней части интерфейса.

Если правильно выстроить логирование в приложении, то можно и без отладчика разобраться, что происходит в коде.

Итого

Приостановить выполнение скрипта можно тремя способами:

1. Точками останова.
2. Использованием в коде команды `debugger`.
3. При ошибке (если инструменты разработчика открыты и опция
включена).

При остановке мы можем отлаживать – проанализировать переменные и пошагово пройти по процессу, что поможет отыскать проблему.

Нами описаны далеко не все инструменты разработчика. С полным руководством можно ознакомиться здесь: https://developers.google.com/web/tools/chrome-devtools.

Для простой отладки вполне достаточно сведений из этой главы, но в дальнейшем рекомендуем вам изучить официальное руководство, если вы собираетесь разрабатывать для браузеров.

И, конечно, вы можете просто покликать в разных местах инструментов разработчика. Пожалуй, это наискорейший способ ими овладеть. Не забывайте про правый клик мыши и контекстные меню!

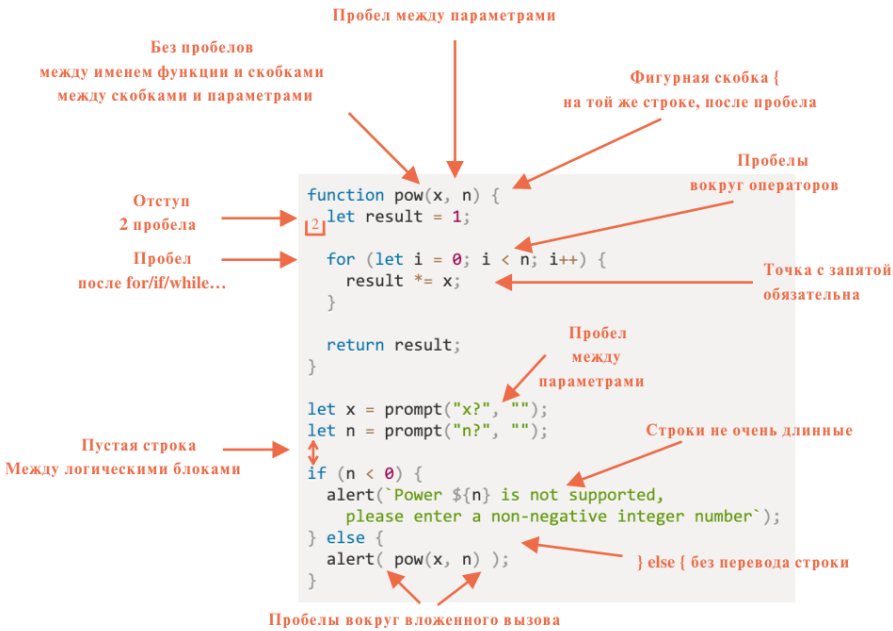
Советы по стилю кода

Код должен быть максимально читаемым и понятным.

Это и есть искусство программирования – взять сложную задачу и написать такой код для её решения, который и правильно работает, и легко читается, понятен для людей. Для этого нужен *хороший стиль* написания кода. В этой главе мы рассмотрим составляющие такого стиля.

Синтаксис

Шпаргалка с правилами синтаксиса (подробнее смотрите ниже по тексту):



Не всё здесь однозначно, так что разберём эти правила подробнее.

> **Ни одно правило не является жёстко обязательным**
>
> Здесь нет железных правил. Это стилевые предпочтения, а не религиозные догмы.

Фигурные скобки

В большинстве JavaScript проектов фигурные скобки пишутся в так называемом «египетском» стиле с открывающей скобкой на той же строке, что и соответствующее ключевое слово – не на новой строке. Перед открывающей скобкой должен быть пробел, как

здесь:

```
if (condition) {
  // делай это
  // ...и это
  // ...и потом это
}
```

А что если у нас однострочная запись, типа `if (condition) doSomething()`, должны ли мы использовать фигурные скобки?

Вот различные варианты расстановки скобок с комментариями, посмотрите сами, какой вам кажется самым читаемым:

1. 😠 Такое иногда бывает в коде начинающих. Плохо, фигурные скобки не нужны:

   ```
   if (n < 0) alert(`Степень ${n} не поддерживается`);
   {
   }
   ```

2. 😠 Никогда не разделяйте строки без фигурных скобок, можно ненароком сделать ошибку при добавлении строк:

   ```
   if (n < 0)
     alert(`Степень ${n} не поддерживается`);
   ```

3. 😏 В одну строку без скобок – приемлемо, если эта строка короткая:

   ```
   if (n < 0) alert(`Степень ${n} не поддерживается`);
   ```

4. 😃 Самый лучший вариант:

   ```
   if (n < 0) {
   ```

```
    alert(`Степень ${n} не поддерживается`);
  }
```

Для очень короткого кода допустима одна строка. Например: `if (cond) return null`. Но блок кода (последний вариант) обычно всё равно читается лучше.

Длина строки

Никто не любит читать длинные горизонтальные строки кода. Лучше всего разбивать их, например:

```
// обратные кавычки ` позволяют разделять строку на части
let str = `
  Рабочая группа TC39 организации Ecma International -
  это группа JavaScript-разработчиков, теоретиков и авторов движков JavaScript,
  которые вместе с сообществом занимаются поддержкой и развитием языка JavaScript.
`;
```

Или для if:

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

Максимальную длину строки согласовывают в команде. Обычно это `80` или `120` символов.

Отступы

Существует два типа отступов:

- **Горизонтальные отступы: два или четыре пробела.**

  Горизонтальный отступ выполняется с помощью 2 или 4 пробелов, или символа табуляции (клавиша `Tab`). Какой из них выбрать – это уже на ваше усмотрение. Пробелы больше распространены.

Одно из преимуществ пробелов над табуляцией заключается в том, что пробелы допускают более гибкие конфигурации отступов, чем символ табуляции.

Например, мы можем выровнять аргументы относительно открывающей скобки:

```
show(parameters,
     aligned, // 5 пробелов слева
     one,
     after,
     another
  ) {
  // ...
}
```

Вертикальные отступы: пустые строки для разбивки кода на «логические блоки».

Даже одну функцию часто можно разделить на логические блоки. В примере ниже разделены инициализация переменных, основной цикл и возвращаемый результат:

```
function pow(x, n) {
  let result = 1;
  //              <--
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //              <--
  return result;
}
```

Вставляйте дополнительный перевод строки туда, где это сделает код более читаемым. Не должно быть более 9 строк кода подряд без вертикального отступа.

Точка с запятой

Точки с запятой должны присутствовать после каждого выражения, даже если их, казалось бы, можно пропустить.

Есть языки, в которых точка с запятой необязательна и редко используется. Однако в JavaScript бывают случаи, когда перенос строки не интерпретируется, как точка с запятой, что может привести к ошибкам. Подробнее об этом – в главе о структуре кода.

Если вы – опытный разработчик на JavaScript, то можно выбрать стиль кода без точек с запятой, например StandardJS. В ином случае, лучше будет использовать точки с запятой, чтобы избежать подводных камней. Большинство разработчиков их ставят.

Уровни вложенности

Уровней вложенности должно быть немного.

Например, в цикле бывает полезно использовать директиву `continue`, чтобы избежать лишней вложенности.

Например, вместо добавления вложенного условия `if`, как здесь:

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- ещё один уровень вложенности
  }
}
```

Мы можем написать:

```
for (let i = 0; i < 10; i++) {
  if (!cond) ;
  ...  // <- нет лишнего уровня вложенности
}
continue
```

Аналогичная ситуация – с `if/else` и `return`.

Например, две нижеследующие конструкции идентичны.

Первая:

```
function pow(x, n) {
  if (n < 0) {
    alert("Отрицательные значения 'n' не поддерживаются");
  } else {
    let result = 1;

    for (let i = 0; i < n; i++) {
      result *= x;
    }

    return result;
  }
}
```

Вторая:

```
function pow(x, n) {
  if (n < 0) {
    alert("Отрицательные значения 'n' не поддерживаются");
    return;
  }

  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Второй вариант является более читабельным, потому что «особый случай» `n < 0` обрабатывается на ранней стадии. После проверки можно переходить к «основному» потоку кода без необходимости увеличения вложенности.

Размещение функций

Если вы пишете несколько вспомогательных функций, а затем используемый ими код, то существует три способа организации функций.

1. Объявить функции *перед* кодом, который их вызовет:

```
//
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}

//
let elem = createElement();
setHandler(elem);
walkAround();
```

объявление функций

код, который использует их

2. Сначала код, затем функции

```
//
let elem = createElement();
setHandler(elem);
walkAround();

// ---  ---
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}
```

код, использующий функции

вспомогательные функции

3. Смешанный стиль: функция объявляется там, где она используется впервые.

В большинстве случаев второй вариант является

предпочтительным.

Это потому, что при чтении кода мы сначала хотим знать, *что он делает*. Если сначала идёт код, то это тут же становится понятно. И тогда, может быть, нам вообще не нужно будет читать функции, особенно если их имена хорошо подобраны.

Руководства по стилю кода

Руководство по стилю содержит общие правила о том, как писать код, например: какие кавычки использовать, сколько пробелов отступать, максимальную длину строки и так далее – в общем, множество мелочей.

Когда все участники команды используют одно и то же руководство по стилю, код выглядит одинаково, независимо от того, кто из команды его написал.

Конечно, команда всегда может написать собственное руководство по стилю, но обычно в этом нет необходимости. Существует множество уже готовых.

Некоторые популярные руководства:

- Google JavaScript Style Guide
- Airbnb JavaScript Style Guide (есть перевод)
- Idiomatic.JS (есть перевод)
- StandardJS
- (и ещё множество других)

Если вы – начинающий разработчик, то начните со шпаргалки в начале этой главы. Как только вы освоитесь, просмотрите другие руководства, чтобы выбрать общие принципы и решить, какое вам больше подходит.

Автоматизированные средства проверки (линтеры)

Автоматизированные средства проверки, так называемые «линтеры» – это инструменты, которые могут автоматически проверять стиль вашего кода и вносить предложения по его улучшению.

Самое замечательное в них то, что проверка стиля может также найти программные ошибки, такие как опечатки в именах переменных или функций. Из-за этой особенности использовать линтер рекомендуется, даже если вы не хотите придерживаться какого-то конкретного «стиля кода».

Вот некоторые известные инструменты для проверки:

JSLint – проверяет код на соответствие стилю JSLint, в онлайн-интерфейсе вверху можно ввести код, а внизу – различные настройки проверки, чтобы попробовать её в действии.

JSHint – больше проверок, чем в JSLint.

ESLint – пожалуй, самый современный линтер.

Все они, в общем-то, работают. Автор пользуется ESLint.

Большинство линтеров интегрированы со многими популярными редакторами: просто включите плагин в редакторе и настройте стиль.

Например, для ESLint вы должны выполнить следующее:

1. Установите Node.JS.
2. Установите ESLint с помощью команды `npm install -g eslint` (npm – установщик пакетов JavaScript).
3. Создайте файл конфигурации с именем `.eslintrc` в корне вашего JavaScript-проекта (в папке, содержащей все ваши файлы).

4. Установите/включите плагин для вашего редактора, который интегрируется с ESLint. У большинства редакторов он есть.

Вот пример файла `.eslintrc`:

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "no-console": 0,
    "indent": ["warning", 2]
  }
}
```

Здесь директива `"extends"` означает, что конфигурация основана на наборе настроек «eslint:recommended». После этого мы уточняем наши собственные.

Кроме того, возможно загрузить наборы правил стиля из сети и расширить их. Смотрите https://eslint.org/docs/user-guide/getting-started для получения более подробной информации об установке.

Также некоторые среды разработки имеют встроенные линтеры, возможно, удобные, но не такие гибкие в настройке, как ESLint.

Итого

Все правила синтаксиса, описанные в этой главе (и в ссылках на руководства по стилю), направлены на повышение читаемости вашего кода. О любых можно поспорить.

Когда мы думаем о написании «лучшего» кода, мы должны

задать себе вопросы: «Что сделает код более читаемым и лёгким для понимания?» и «Что может помочь избегать ошибок?». Это – основные моменты, о которых следует помнить при выборе и обсуждении стилей кода.

Чтение популярных руководств по стилю позволит вам быть в курсе лучших практик и последних идей и тенденций в стилях написания кода.

Задачи

Плохой стиль

важность: 4

Какие недостатки вы видите в стиле написания кода этого примера?

```
function pow(x,n)
{
  let result=1;
  for(let i=0;i<n;i++) {result*=x;}
  return result;
}

let x=prompt("x?",''), n=prompt("n?",'')
if (n<=0)
{
  alert(`Степень ${n} не поддерживается, введите целую
степень, большую 0`);
}
else
{
  alert(pow(x,n))
}
```

К решению

Комментарии

Как мы знаем из главы Структура кода, комментарии могут быть однострочными, начинающимися с `//`, и многострочными: `/* ... */`.

Обычно мы их используем, чтобы описать, как и почему работает код.

На первый взгляд, в комментариях нет ничего сложного, но новички в программировании часто применяют их неправильно.

Плохие комментарии

Новички склонны использовать комментарии, чтобы объяснять, «что делает код». Например, так:

```
// Этот код делает это (...) и вот это (...)
// ...и кто знает, что ещё...
очень;
сложный;
код;
```

Но в хорошем коде количество «объясняющих» комментариев должно быть минимальным. Серьёзно, код должен быть таким, чтобы его можно было понять без комментариев.

Про это есть хорошее правило: «Если код настолько запутанный, что требует комментариев, то, может быть, его стоит переделать?»

Рецепт: выносите код в функции

Иногда выгодно заменить часть кода функцией, например, в таком случае:

```
function showPrimes(n) {
```

```
  nextPrime:
  for (let i = 2; i < n; i++) {



    alert(i);
  }
}
     // проверяем, является ли i простым числом
     for (let j = 2; j < i; j++) {
       if (i % j == 0) continue nextPrime;
     }
```

Лучший вариант – использовать отдельную функцию `isPrime`:

```
function showPrimes(n) {

  for (let i = 2; i < n; i++) {


    alert(i);
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if (n % i == 0) return false;
  }

  return true;
}
if
  (!isPrime(i)) continue;
```

Теперь код легче понять. Функция сама становится комментарием. Такой код называется *самодокумен т ированным*.

Рецепт: создавайте функции

И если мы имеем такой длинный кусок кода:

```
// здесь мы добавляем виски
for(let i = 0; i < 10; i++) {
```

```
  let drop = getWhiskey();
  smell(drop);
  add(drop, glass);
}

// здесь мы добавляем сок
for(let t = 0; t < 3; t++) {
  let tomato = getTomato();
  examine(tomato);
  let juice = press(tomato);
  add(juice, glass);
}

// ...
```

То будет лучше отрефакторить его с использованием функций:

```
addWhiskey(glass);
addJuice(glass);

function addWhiskey(container) {
  for(let i = 0; i < 10; i++) {
    let drop = getWhiskey();
    //...
  }
}

function addJuice(container) {
  for(let t = 0; t < 3; t++) {
    let tomato = getTomato();
    //...
  }
}
```

Здесь комментарии тоже не нужны: функции сами говорят, что делают (если вы понимаете английский язык). И ещё, структура кода лучше, когда он разделён на части. Понятно, что делает каждая функция, что она принимает и что возвращает.

В реальности мы не можем полностью избежать «объясняющих» комментариев. Существуют сложные алгоритмы. И есть хитрые уловки для оптимизации. Но в целом мы должны стараться писать простой и

самодокументированный код.

Хорошие комментарии

Итак, обычно «объясняющие» комментарии – это плохо. Но тогда какой комментарий считается хорошим?

Описывайте архитектуру

Сделайте высокоуровневый обзор компонентов, того, как они взаимодействуют, каков поток управления в различных ситуациях… Если вкратце – обзор кода с высоты птичьего полёта. Существует специальный язык UML для создания диаграмм, разъясняющих архитектуру кода. Его определённо стоит изучить.

Документируйте параметры и использование функций

Есть специальный синтаксис JSDoc для документирования функций: использование, параметры, возвращаемое значение.

Например:

```
/**
 * Возвращает x, возведённое в n-ную степень.
 *
 * @param {number} x Возводимое в степень число.
 * @param {number} n Степень, должна быть натуральным числом.
 * @return {number} x, возведённое в n-ную степень.
 */
function pow(x, n) {
  ...
}
```

Подобные комментарии позволяют нам понимать назначение функции и правильно её использовать без необходимости заглядывать в код.

Кстати, многие редакторы, такие как WebStorm, прекрасно

их распознают для того, чтобы выполнить автодополнение ввода и различные автоматические проверки кода.

Также существуют инструменты, например, JSDoc 3, которые умеют генерировать HTML-документацию из комментариев. Получить больше информации о JSDoc вы можете здесь: http://usejsdoc.org/.

Почему задача решена именно таким способом?

Важно то, что написано. Но то, что *не* написано, может быть даже более важным, чтобы понимать происходящее. Почему задача решена именно этим способом? Код не даёт ответа.

Если есть несколько способов решить задачу, то почему вы выбрали именно этот? Особенно если ваш способ – не самый очевидный.

Без подобных комментариев возможна следующая ситуация:

1. Вы (или ваш коллега) открываете написанный некоторое время назад код и видите, что в нём есть, что улучшить.
2. Вы думаете: «Каким глупым я раньше был и насколько умнее стал сейчас», и переписываете его на «более правильный и оптимальный» вариант.
3. …Желание переписать код – это хорошо. Но в процессе вы понимаете, что «оптимальное» решение на самом деле не такое уж и оптимальное. Вы даже смутно припоминаете, почему, так как в прошлый раз вы уже его пробовали. Вы возвращаетесь к правильному варианту, потратив время зря.

Комментарии, объясняющие решение, очень важны. Они помогают продолжать разработку в правильном направлении.

В коде есть какие-то тонкости? Где они используются?

Если в коде есть какие-то тонкости и неочевидные вещи, его определённо нужно комментировать.

Итого

Комментарии – важный признак хорошего разработчика, причём как их наличие, так и отсутствие.

Хорошие комментарии позволяют нам поддерживать код, дают возможность вернуться к нему после перерыва и эффективнее его использовать.

Комментируйте:

- Общую архитектуру, вид «с высоты птичьего полёта».
- Использование функций.
- Неочевидные решения, важные детали.

Избегайте комментариев:

- Которые объясняют, как работает код, и что он делает.
- Используйте их только в тех случаях, когда невозможно сделать настолько простой и самодокументированный код, что он не потребует комментариев.

Средства для генерации документации по коду, такие как JSDoc3, также используют комментарии: они их читают и генерируют HTML-документацию (или документацию в другом формате).

Ниндзя-код

Предлагаю вашему вниманию советы мастеров древности.

Программисты прошлого использовали их, чтобы заострить разум тех, кто после них будет поддерживать код.

Гуру разработки при найме старательно ищут их применение в тестовых заданиях.

Новички иногда используют их ещё лучше, чем матёрые ниндзя.

Прочитайте их и решите, кто вы: ниндзя, новичок или, может быть, гуру?

> **Осторожно, ирония!**
>
> Многие пытались пройти по пути ниндзя. Мало, кто преуспел.

Краткость – сестра таланта!

Пишите «как короче», а не как понятнее. Покажите, насколько вы умны!

«Меньше букв» – уважительная причина для нарушения любых соглашений. Ваш верный помощник – возможности языка, использованные неочевидным образом.

Обратите внимание на оператор вопросительный знак `'?'`, например:

```
// код из jQuery
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Разработчик, встретивший эту строку и попытавшийся

понять, чему же всё-таки равно `i`, скорее всего, придёт к вам за разъяснениями. Смело скажите ему, что короче – это всегда лучше. Посвятите и его в пути ниндзя. Не забудьте вручить Дао дэ цзин.

Однобуквенные переменные

> *Кто знает — не говорит.*
> *Кто говорит — не знает.*
> Лао-цзы

Ещё один способ писать быстрее – использовать короткие имена переменных. Называйте их `a`, `b` или `c`.

Короткая переменная прячется в коде лучше, чем ниндзя в лесу. Никто не сможет найти её, используя функцию «Поиск» текстового редактора. Более того, даже найдя – никто не сможет «расшифровать» её и догадаться, что она означает.

…Но есть одно исключение. В тех местах, где однобуквенные переменные общеприняты, например, в счётчике цикла – ни в коем случае не используйте стандартные названия `i`, `j`, `k`. Где угодно, только не здесь!

Остановите свой взыскательный взгляд на чём-нибудь более экзотическом. Например, `x` или `y`.

Эффективность этого подхода особенно заметна, если тело цикла занимает одну-две страницы (чем длиннее – тем лучше).

В этом случае заметить, что переменная – счётчик цикла, без пролистывания вверх, невозможно.

Используйте сокращения

Если правила, принятые в вашей команде, запрещают использовать абстрактные имена или имена из одной буквы – сокращайте их.

Например:

`list` → `lst`.

`userAgent` → `ua`.

`browser` → `brsr`.

…и т.д.

Только коллеги с хорошо развитой интуицией поймут такие имена. Вообще, старайтесь сокращать всё. Только одарённые интуицией люди достойны заниматься поддержкой вашего кода.

Будьте абстрактны при выборе имени.

> *Лучший кувшин лепят всю жизнь,*
> *Высокая музыка неподвластна слуху,*
> *Великий образ не имеет формы.*
> Лао-цзы

При выборе имени старайтесь применить максимально абстрактное слово, например `obj`, `data`, `value`, `item`, `elem` и т.п.

Идеальное имя для переменной: `data`. Используйте

это имя везде, где можно. В конце концов, каждая переменная содержит *данные*, не правда ли?

…Но что делать, если имя `data` уже занято? Попробуйте `value`, оно не менее универсально. Ведь каждая переменная содержит *значение*.

Называйте переменную по типу данных, которые она хранит: `str`, `num`...

Попробуйте! Сделают ли такие имена интереснее разработку? Как ни странно, да и намного!

Казалось бы, название переменной содержит информацию, говорит о том, что в переменной – число, объект или массив… С другой стороны, **когда непосвящённый будет разбирать этот код – он с удивлением обнаружит, что информации нет!**

Ведь как раз тип легко понять, запустив отладчик и посмотрев, что внутри. Но в чём смысл этой переменной? Что за массив/объект/число в ней хранится? Без долгой медитации над кодом тут не обойтись!

…Но что делать, если и эти имена закончились? Просто добавьте цифру: `data1`, `item2`, `elem5`...

Проверка внимания

Только истинно внимательный программист достоин понять ваш код. Но как проверить, достоин ли читающий?

Один из способов – использовать похожие имена переменных, например, `date` и `data`.

Бегло прочитать такой код почти невозможно. А уж заметить опечатку и поправить её… Мммммм… Мы здесь надолго, время попить чайку.

Русские слова и сокращения

Если вам *приходи т ся* использовать длинные, понятные имена переменных – что поделать… Но и здесь есть простор для творчества!

Назовите переменные «калькой» с русского языка или как-то «улучшите» английское слово.

В одном месте напишите `var ssilka`, в другом `var ssylka`, в третьем `var link`, в четвёртом – `var lnk`… Это действительно великолепно работает и очень креативно!

Количество ошибок при поддержке такого кода увеличивается во много раз.

Хитрые синонимы

> *Очень трудно найти чёрную кошку в тёмной комнате, особенно, когда её там нет.*
> Конфуций

Чтобы было не скучно – используйте *похожие* названия для обозначения *одинаковых* действий.

Например, если метод показывает что-то на экране – начните его название с `display..` (скажем, `displayElement`), а в другом месте объявите аналогичный метод как `show..` (`showFrame`).

Как бы намекните этим, что существует тонкое различие между способами показа в этих методах,

хотя на самом деле его нет.

По возможности, договоритесь с членами своей команды. Если Вася в своих классах использует `display..`, то Валера – обязательно `render..`, а Петя – `paint...`

…И напротив, если есть две функции с важными отличиями – используйте одно и то же слово для их описания! Например, с `print...` можно начать метод печати на принтере `printPage`, а также – метод добавления текста на страницу `printText`.

А теперь пусть читающий код думает: «Куда же выводит сообщение printMessage?». Особый шик – добавить элемент неожиданности. Пусть `printMessage` выводит не туда, куда все, а в новое окно!

Повторно используйте имена

> *Когда целое разделено, его частям*
> *нужны имена.*
> *Уже достаточно имён.*
> *Нужно знать, когда остановиться.*
> Лао-цзы

По возможности, повторно используйте имена переменных, функций и свойств. Просто записывайте в них новые значения.

Добавляйте новое имя, только если это абсолютно необходимо. В функции старайтесь обойтись только теми переменными, которые были переданы как параметры.

Это не только затруднит идентификацию того, что *сейчас* находится в переменной, но и сделает почти невозможным поиск места, в котором конкретное значение было присвоено.

Цель – развить интуицию и память читающего код программиста. Ну, а пока интуиция слаба, он может построчно анализировать код и конспектировать изменения переменных для каждой ветки исполнения.

Продвинутый вариант этого подхода – незаметно (!) подменить переменную на нечто похожее, например:

```
function ninjaFunction(elem) {
  // 20 строк кода, работающего с elem

  elem = clone(elem);

  // ещё 20 строк кода, работающего с elem!
}
```

Программист, пожелавший добавить действия с `elem` во вторую часть функции, будет удивлён. Лишь во время отладки, посмотрев весь код, он с удивлением обнаружит, что, оказывается, имел дело с клоном!

Регулярные встречи с этим приёмом на практике говорят: защититься невозможно. Эффективно даже против опытного ниндзи.

Добавляйте подчёркивания

Добавляйте подчёркивания `_` и `__` к именам переменных. Например, `_name` или `__value`. Желательно, чтобы их смысл был известен только вам, а лучше – вообще без явной причины.

Этим вы достигните двух целей. Во-первых, код станет длиннее и менее читаемым, а во-вторых, другой

программист будет долго искать смысл в подчёркиваниях. Особенно хорошо сработает и внесёт сумятицу в его мысли, если в некоторых частях проекта подчёркивания будут, а в некоторых – нет.

В процессе развития кода вы, скорее всего, будете путаться и смешивать стили: добавлять имена с подчёркиваниями там, где обычно подчёркиваний нет, и наоборот. Это нормально и полностью соответствует третьей цели – увеличить количество ошибок при внесении исправлений.

Покажите вашу любовь к разработке

Пусть все видят, какими замечательными сущностями вы оперируете! Имена `superElement`, `megaFrame` и `niceItem` при благоприятном положении звёзд могут привести к просветлению читающего.

Действительно, с одной стороны, кое-что написано: `super..`, `mega..`, `nice..` С другой – это не несёт никакой конкретики. Читающий может решить поискать в этом глубинный смысл и замедитировать на часок-другой оплаченного рабочего времени.

Перекрывайте внешние переменные

> *Находясь на свету, нельзя*
> *ничего увидеть в темноте.*
> *Пребывая же в темноте,*
> *увидишь все, что находится*
> *на свету.*
> Гуань
> Инь-цзы

Почему бы не использовать одинаковые переменные внутри и снаружи функции? Это просто и не требует придумывать новых имён.

```
let  = authenticateUser();

function render() {
  let  = anotherValue();
  ...
  ...многобукв...
  ...
  ... // <-- программист захочет внести исправления сюда, и...
  ...
}
```

```
user
```

```
user
```

Зашедший в середину метода `render` программист, скорее всего, не заметит, что переменная `user` локально перекрыта и попытается работать с ней, полагая, что это – результат `authenticateUser()`… Ловушка захлопнулась! Здравствуй, отладчик.

Внимание… Сюр-при-из!

Есть функции, название которых говорит о том, что они ничего не меняют. Например, `isReady()`, `checkPermission()`, `findTags()`… Предполагается, что при вызове они произведут некие вычисления или найдут и возвратят полезные данные, но при этом их не изменят. В трактатах это называется «отсутствие сторонних эффектов».

По-настоящему красивый приём – делать в таких функциях что-нибудь полезное, заодно с процессом проверки. Что именно – совершенно неважно.

Удивление и ошеломление, которое возникнет у вашего коллеги, когда он увидит, что функция с названием на `is..`, `check..` или `find...` что-то меняет –

несомненно, расширит его границы разумного!

Ещё одна вариация такого подхода – возвращать нестандартное значение.

Ведь общеизвестно, что is… и check… обычно возвращают true/false. Продемонстрируйте оригинальное мышление. Пусть вызов `checkPermission` возвращает не результат `true/false`, а объект с результатами проверки! А что, полезно.

Те же разработчики, кто попытается написать проверку `if (checkPermission(..))`, будут весьма удивлены результатом. Ответьте им: «Надо читать документацию!». И перешлите эту статью.

Мощные функции!

> *Дао везде и во всём,*
> *и справа, и слева.*
> Лао-цзы

Не ограничивайте действия функции тем, что написано в её названии. Будьте шире.

Например, функция `validateEmail(email)` может, кроме проверки e-mail на правильность, выводить сообщение об ошибке и просить заново ввести e-mail.

Выберите хотя бы пару дополнительных действий, кроме основного назначения функции. Главное – они должны быть неочевидны из названия функции. Истинный ниндзя-разработчик сделает так, что они будут неочевидны и из кода тоже.

Объединение нескольких смежных действий в одну функцию защитит ваш код от повторного использования.

Представьте, что другому разработчику нужно только проверить адрес, а сообщение – не выводить. Ваша функция `validateEmail(email)`, которая делает и то и другое, ему не подойдёт. И он не прервёт вашу медитацию вопросами о ней.

Итого

Все советы выше пришли из реального кода… И в том числе, от разработчиков с большим опытом. Возможно, даже больше вашего, так что не судите опрометчиво ;)

Следуйте нескольким из них – и ваш код станет полон сюрпризов.

Следуйте многим – и ваш код станет истинно вашим, никто не захочет изменять его.

Следуйте всем – и ваш код станет ценным уроком для молодых разработчиков, ищущих просветления.

Автоматическое тестирование с использованием фреймворка Mocha

Далее у нас будут задачи, для проверки которых используется автоматическое тестирование. Также его часто применяют в реальных проектах.

Зачем нам нужны тесты?

Обычно, когда мы пишем функцию, мы легко можем представить, что она должна делать, и как она будет вести

себя в зависимости от переданных параметров.

Во время разработки мы можем проверить правильность работы функции, просто вызвав её, например, из консоли и сравнив полученный результат с ожидаемым.

Если функция работает не так, как мы ожидаем, то можно внести исправления в код и запустить её ещё раз. Так можно повторять до тех пор, пока функция не станет работать так, как нам нужно.

Однако, такие «ручные перезапуски» – не лучшее решение.

При тестировании кода ручными перезапусками легко упустить что-нибудь важное.

Например, мы работаем над функцией `f`. Написали часть кода и решили протестировать. Выясняется, что `f(1)` работает правильно, в то время как `f(2)` – нет. Мы вносим в код исправления, и теперь `f(2)` работает правильно. Вроде бы, всё хорошо, не так ли? Однако, мы забыли заново протестировать `f(1)`. Возможно, после внесения правок `f(1)` стала работать неправильно.

Это – типичная ситуация. Во время разработки мы учитываем множество различных сценариев использования. Но сложно ожидать, что программист станет вручную проверять каждый из них после любого изменения кода. Поэтому легко исправить что-то одно и при этом сломать что-то другое.

Автоматическое тестирование означает, что тесты пишутся отдельно, в дополнение к коду. Они по-разному запускают наши функции и сравнивают результат с ожидаемым.

Behavior Driven Development (BDD)

Давайте начнём с техники под названием Behavior Driven Development или, коротко, BDD.

BDD – это три в одном: и тесты, и документация, и примеры использования.

Чтобы понять BDD – рассмотрим практический пример разработки.

Разработка функции возведения в степень — «pow»: спецификация

Допустим, мы хотим написать функцию `pow(x, n)`, которая возводит `x` в целочисленную степень `n`. Мы предполагаем, что `n≥0`.

Эта задача взята в качестве примера. В JavaScript есть оператор `**`, который служит для возведения в степень. Мы сосредоточимся на процессе разработки, который также можно применять и для более сложных задач.

Перед тем, как начать писать код функции `pow`, мы можем представить себе, что она должна делать, и описать её.

Такое описание называется *спецификацией* (specification). Она содержит описания различных способов использования и тесты для них, например:

```
describe("pow", function() {

  it("возводит в степень n", function() {
    assert.equal(pow(2, 3), 8);
  });

});
```

Спецификация состоит из трёх основных блоков:

`describe("заголовок", function() { ... })`

Какую функциональность мы описываем. В нашем случае мы описываем функцию `pow`. Используется для группировки рабочих лошадок – блоков `it`.

```
it("описание", function() { ... })
```

В первом аргументе блока `it` мы *человеческим языком* описываем конкретный способ использования функции, а во втором – пишем функцию, которая тестирует данный случай.

```
assert.equal(value1, value2)
```

Код внутри блока `it`, если функция работает верно, должен выполняться без ошибок.

Функции вида `assert.*` используются для проверки того, что функция `pow` работает так, как мы ожидаем. В этом примере мы используем одну из них – `assert.equal`, которая сравнивает переданные значения и выбрасывает ошибку, если они не равны друг другу. Существуют и другие типы сравнений и проверок, которые мы добавим позже.

Спецификация может быть запущена, и при этом будет выполнена проверка, указанная в блоке `it`, мы увидим это позднее.

Процесс разработки

Процесс разработки обычно выглядит следующим образом:

1. Пишется начальная спецификация с тестами, проверяющими основную функциональность.
2. Создаётся начальная реализация.
3. Для запуска тестов мы используем фреймворк Mocha

(подробнее о нём чуть позже). Пока функция не готова, будут ошибки. Вносим изменения до тех пор, пока всё не начнёт работать так, как нам нужно.

4. Теперь у нас есть правильно работающая начальная реализация и тесты.
5. Мы добавляем новые способы использования в спецификацию, возможно, ещё не реализованные в тестируемом коде. Тесты начинают «падать» (выдавать ошибки).
6. Возвращаемся на шаг 3, дописываем реализацию до тех пор, пока тесты не начнут завершаться без ошибок.
7. Повторяем шаги 3-6, пока требуемая функциональность не будет готова.

Таким образом, разработка проходит *итеративно*. Мы пишем спецификацию, реализуем её, проверяем, что тесты выполняются без ошибок, пишем ещё тесты, снова проверяем, что они проходят и т.д.

Давайте посмотрим этот поток разработки на нашем примере.

Первый шаг уже завершён. У нас есть спецификация для функции `pow`. Теперь, перед тем, как писать реализацию, давайте подключим библиотеки для пробного запуска тестов, просто чтобы убедиться, что тесты работают (разумеется, они завершатся ошибками).

Спецификация в действии

В этой главе мы будем пользоваться следующими JavaScript-библиотеками для тестов:

Mocha – основной фреймворк. Он предоставляет общие функции тестирования, такие как `describe` и `it`, а также функцию запуска тестов.

Chai – библиотека, предоставляющая множество функций проверки утверждений. Пока мы будем использовать только `assert.equal`.

Sinon – библиотека, позволяющая наблюдать за функциями, эмулировать встроенные функции и многое другое. Нам она пригодится позднее.

Эти библиотеки подходят как для тестирования внутри браузера, так и на стороне сервера. Мы рассмотрим вариант с браузером.

Полная HTML-страница с этими библиотеками и спецификацией функции `pow`:

```
<!DOCTYPE html>
<html>
<head>
  <!-- добавим стили mocha для отображения результатов -->
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/
ajax/libs/mocha/3.2.0/mocha.css">
  <!-- добавляем сам фреймворк mocha -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/
mocha/3.2.0/mocha.js"></script>
  <script>
    // включаем режим тестирования в стиле BDD
    mocha.setup('bdd');
  </script>
  <!-- добавим chai -->
  <script src="https://cdnjs.cloudflare.com/ajax/libs/
chai/3.5.0/chai.js"></script>
  <script>
    // chai предоставляет большое количество функций. Объявим
assert глобально
    let assert = chai.assert;
  </script>
</head>

<body>

  <script>
    function pow(x, n) {
      /* Здесь будет реализация функции, пока пусто */
    }
  </script>

  <!-- скрипт со спецификацией (describe, it...) -->
```

```
  <script src="test.js"></script>

  <!-- элемент с id="mocha" будет содержать результаты тестов
-->
  <div id="mocha"></div>

  <!-- запускаем тесты! -->
  <script>
    mocha.run();
  </script>
</body>

</html>
```

Условно страницу можно разделить на пять частей:

1. Тег `<head>` содержит сторонние библиотеки и стили для тестов.
2. Тег `<script>` содержит тестируемую функцию, в нашем случае – `pow`.
3. Тесты – в нашем случае внешний скрипт `test.js`, который содержит спецификацию `describe("pow", ...)`, представленную выше.
4. HTML-элемент `<div id="mocha">` будет использован фреймворком Mocha для вывода результатов тестирования.
5. Запуск тестов производится командой `mocha.run()`.

Результаты:

Пока что тест завершается ошибкой. Это логично, потому что у нас пустая функция `pow`, так что `pow(2,3)` возвращает `undefined` вместо `8`.

На будущее отметим, что существуют более высокоуровневые фреймворки для тестирования, такие как karma и другие. С их помощью легко сделать автозапуск множества тестов.

Начальная реализация

Давайте напишем простую реализацию функции `pow`, чтобы пройти тесты.

```
function pow(x, n) {
  return 8; // :) сжульничаем!
}
```

Вау, теперь всё работает!

Улучшаем спецификацию

Конечно, мы сжульничали. Функция не работает. Попытка посчитать `pow(3,4)` даст некорректный результат, однако тесты проходят.

…Такая ситуация вполне типична, она случается на практике. Тесты проходят, но функция работает неправильно. Наша спецификация не идеальна. Нужно дополнить её тестами.

Давайте добавим ещё один тест, чтобы посмотреть, что `pow(3, 4) = 81`.

У нас есть два пути организации тестов:

1. Первый – добавить ещё один `assert` в существующий `it`:

```
describe("pow", function() {

  it("возводит число в степень n", function() {
    assert.equal(pow(2, 3), 8);

  });

});
```

```
assert.equal(pow(3, 4), 81);
```

2. Второй – написать два теста:

```
describe("pow", function() {

  it("2 в степени 3 будет 8", function() {
    assert.equal(pow(2, 3), 8);
  });

  it("3 в степени 4 будет 81", function() {
    assert.equal(pow(3, 4), 81);
  });

});
```

Принципиальная разница в том, что когда один из `assert` выбрасывает ошибку, то выполнение `it` блока тут же прекращается. Таким образом, если первый `assert` выбросит ошибку, результат работы второго `assert` мы уже не узнаем.

Разделять тесты предпочтительнее, так как мы получаем больше информации о том, что конкретно пошло не так.

Помимо этого есть одно хорошее правило, которому стоит следовать.

Один тест проверяет одну вещь.

Если вы посмотрите на тест и увидите в нём две независимые проверки, то такой тест лучше разделить на два более простых.

Давайте продолжим со вторым вариантом.

Результаты:

Как мы и ожидали, второй тест провалился. Естественно, наша функция всегда возвращает `8`, в то время как `assert` ожидает `81`.

Улучшаем реализацию

Давайте напишем что-то более похожее на функцию возведения в степень, чтобы заставить тесты проходить.

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Чтобы убедиться, что эта реализация работает нормально, давайте протестируем её на большем количестве значений. Чтобы не писать вручную каждый блок `it`, мы можем генерировать их в цикле `for`:

```
describe("pow", function() {

  function makeTest(x) {
    let expected = x * x * x;
    it(`${x} в степени 3 будет ${expected}`, function() {
      assert.equal(pow(x, 3), expected);
    });
  }

  for (let x = 1; x <= 5; x++) {
    makeTest(x);
  }

});
```

Результат:

Вложенные блоки describe

Мы собираемся добавить больше тестов. Однако, перед этим стоит сгруппировать вспомогательную функцию `makeTest` и цикл `for`. Нам не нужна функция `makeTest` в

других тестах, она нужна только в цикле `for`. Её предназначение – проверить, что `pow` правильно возводит число в заданную степень.

Группировка производится вложенными блоками `describe`:

```
describe("pow", function() {

    function makeTest(x) {
      let expected = x * x * x;
      it(`${x} в степени 3 будет ${expected}`, function() {
        assert.equal(pow(x, 3), expected);
      });
    }

    for (let x = 1; x <= 5; x++) {
      makeTest(x);
    }

  // ... другие тесты. Можно писать и describe, и it блоки.
});
```

```
  describe("возводит x в степень 3", function() {
  });
```

Вложенные `describe` образуют новую подгруппу тестов. В результатах мы можем видеть дополнительные отступы в названиях.

В будущем мы можем написать новые `it` и `describe` блоки на верхнем уровне со своими собственными вспомогательными функциями. Им не будет доступна функция `makeTest` из примера выше.

before/after и beforeEach/afterEach

Мы можем задать `before/after` функции, которые будут выполняться до/после тестов, а также функции `beforeEach/afterEach`, выполняемые до/после *каждого* `it`.

Например:

```
describe("тест", function() {

  before(() => alert("Тестирование началось – перед тестами"));
  after(() => alert("Тестирование закончилось – после всех тестов"));

  beforeEach(() => alert("Перед тестом – начинаем выполнять тест"));
  afterEach(() => alert("После теста – заканчиваем выполнение теста"));

  it('тест 1', () => alert(1));
  it('тест 2', () => alert(2));

});
```

Порядок выполнения будет таким:

```
Тестирование началось – перед тестами (before)
Перед тестом – начинаем выполнять тест (beforeEach)
1
После теста – заканчиваем выполнение теста
(afterEach)
Перед тестом – начинаем выполнять тест (beforeEach)
2
После теста – заканчиваем выполнение теста
(afterEach)
Тестирование закончилось – после всех тестов (after)
```

Открыть пример в песочнице.

Обычно `beforeEach/afterEach` и `before/after` используются для инициализации, обнуления счётчиков или чего-нибудь ещё между тестами (или группами тестов).

Расширение спецификации

Основная функциональность `pow` реализована. Первая итерация разработки завершена. Когда мы закончим отмечать и пить шампанское, давайте продолжим работу и улучшим `pow`.

Как было сказано, функция `pow(x, n)` предназначена для работы с целыми положительными значениями `n`.

Для обозначения математических ошибок функции JavaScript обычно возвращают `NaN`. Давайте делать также для некорректных значений `n`.

Сначала давайте опишем это поведение в спецификации.

```
describe("pow", function() {

  // ...

  it("для отрицательных n возвращает NaN", function() {

  });

  it("для дробных n возвращает NaN", function() {

  });

});
```

```
    assert.isNaN(pow(2, -1));
    assert.isNaN(pow(2, 1.5));
```

Результаты с новыми тестами:

Новые тесты падают, потому что наша реализация не поддерживает их. Так работает BDD. Сначала мы добавляем тесты, которые падают, а уже потом пишем под них реализацию.

Другие функции сравнения

Обратите внимание на `assert.isNaN`. Это проверка того, что переданное значение равно `NaN`.

Библиотека Chai содержит множество других подобных функций, например:

- `assert.equal(value1, value2)` – проверяет равенство `value1 == value2`.
- `assert.strictEqual(value1, value2)` – проверяет строгое равенство `value1 === value2`.
- `assert.notEqual`, `assert.notStrictEqual` – проверяет неравенство и строгое неравенство соответственно.
- `assert.isTrue(value)` – проверяет, что `value === true`
- `assert.isFalse(value)` – проверяет, что `value === false`
- …с полным списком можно ознакомиться в документации

Итак, нам нужно добавить пару строчек в функцию `pow`:

```
function pow(x, n) {


  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
  if (n < 0) return NaN;
  if (Math.round(n) != n) return NaN;
```

Теперь работает, все тесты проходят:

Итого

В BDD сначала пишут спецификацию, а потом реализацию. В конце у нас есть и то, и другое.

Спецификацию можно использовать тремя способами:

1. Как **Тесты** – они гарантируют, что функция работает правильно.
2. Как **Документацию** – заголовки блоков `describe` и `it` описывают поведение функции.
3. Как **Примеры** – тесты, по сути, являются готовыми примерами использования функции.

Имея спецификацию, мы можем улучшить, изменить и даже переписать функцию с нуля, и при этом мы будем уверены, что она продолжает работать правильно.

Это особенно важно в больших проектах, когда одна функция может быть использована во множестве мест. Когда мы вносим в такую функцию изменения, у нас нет никакой возможности вручную проверить, что она продолжает работать правильно во всех местах, где её используют.

Не имея тестов, людям приходится выбирать один из двух путей:

1. Внести изменения, и неважно, что будет. Потом у наших пользователей станут проявляться ошибки, ведь мы наверняка что-то забудем проверить вручную.
2. Или же, если наказание за ошибки в коде серьёзное, то люди просто побоятся вносить изменения в такие функции. Код будет стареть, «зарастать паутиной», и

никто не захочет в него лезть. Это нехорошо для разработки.

Автоматическое тестирование кода позволяет избежать этих проблем!

Если проект покрыт тестами, то вышеупомянутые проблемы не возникают. После любых изменений мы можем запустить тесты и увидеть результаты огромного количества проверок, сделанных за секунды.

Кроме того, код, хорошо покрытый тестами, как правило, имеет лучшую архитектуру.

Это естественно, ведь такой код легче менять и улучшать. Но не только по этой причине.

Для написания тестов нужно организовать код таким образом, чтобы у каждой функции была ясно поставленная задача и точно определены её аргументы и возвращаемое значение. А это означает, что мы получаем хорошую архитектуру с самого начала.

В реальности это не всегда так просто. Иногда сложно написать спецификацию до того, как будет написана реализация, потому что не всегда чётко понятно, как та или иная функция должна себя вести. Но в общем и целом написание тестов делает разработку быстрее, а итоговый продукт более стабильным.

Далее по книге мы встретим много задач с тестами, так что вы увидите много практических примеров.

Написание тестов требует хорошего знания JavaScript. Но мы только начали учить его. Не волнуйтесь. Пока вам не нужно писать тесты, но вы уже умеете их читать и поймёте даже более сложные примеры, чем те, что были представлены в этой главе.

Задачи

Что не так с этим тестом?

важность: 5

Что не так в нижеприведённом тесте функции `pow`?

```
it("Возводит x в степень n", function() {
  let x = 5;

  let result = x;
  assert.equal(pow(x, 1), result);

  result *= x;
  assert.equal(pow(x, 2), result);

  result *= x;
  assert.equal(pow(x, 3), result);
});
```

P.S. Тест не содержит синтаксических ошибок и успешно проходит.

К решению

Полифилы

JavaScript – динамично развивающийся язык программирования. Регулярно появляются предложения о добавлении в JS новых возможностей, они анализируются, и, если предложения одобряются, то описания новых возможностей языка переносятся в черновик https://tc39.github.io/ecma262/, а затем публикуются в спецификации.

Разработчики JavaScript-движков сами решают, какие предложения реализовывать в первую очередь. Они могут заранее добавить в браузеры поддержку функций, которые всё ещё находятся в черновике, и отложить разработку функций, которые уже перенесены в спецификацию, потому что они менее интересны разработчикам или более сложные в реализации.

Таким образом, довольно часто реализуется только часть стандарта.

Можно проверить текущее состояние поддержки различных возможностей JavaScript на странице https://kangax.github.io/compat-table/es6/ (нам ещё предстоит изучить многое из этого списка).

Babel

Когда мы используем современные возможности JavaScript, некоторые движки могут не поддерживать их. Как было сказано выше, не везде реализованы все функции.

И тут приходит на помощь Babel.

Babel – это транспилер. Он переписывает современный JavaScript-код в предыдущий стандарт.

На самом деле, есть две части Babel:

1. Во-первых, транспилер, который переписывает код. Разработчик запускает Babel на своём компьютере. Он переписывает код в старый стандарт. И после этого код отправляется на сайт. Современные сборщики проектов, такие как webpack или brunch, предоставляют возможность запускать транспилер автоматически после каждого изменения кода, что позволяет экономить время.

2. Во-вторых, полифил.

   Новые возможности языка могут включать встроенные функции и синтаксические конструкции. Транспилер переписывает код, преобразовывая новые синтаксические конструкции в старые. Но что касается новых встроенных функций, нам нужно их как-то реализовать. JavaScript является высокодинамичным языком, скрипты могут добавлять/изменять любые функции, чтобы они вели себя в соответствии с современным стандартом.

   Термин «полифил» означает, что скрипт «заполняет» пробелы и добавляет современные функции.

   Два интересных хранилища полифилов:

   core js поддерживает много функций, можно подключать только нужные.

   polyfill.io – сервис, который автоматически создаёт скрипт с полифилом в зависимости от необходимых функций и браузера пользователя.

Таким образом, чтобы современные функции поддерживались в старых движках, нам надо установить транспилер и добавить полифил.

Примеры в учебнике

Вы читаете оффлайн-версию, примеры в PDF запустить не получится, в EPUB некоторые работают.

Google Chrome обычно поддерживает современные функции, можно запускать новейшие примеры без каких-либо транспилеров, но и другие современные браузеры тоже хорошо работают.

Объекты: основы

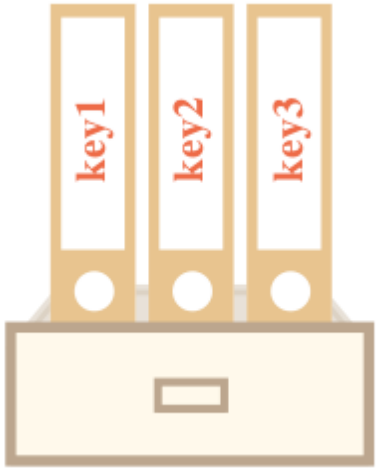
Объекты

Как мы знаем из главы Типы данных, в JavaScript существует 8 типов данных. Семь из них называются «примитивными», так как содержат только одно значение (будь то строка, число или что-то другое).

Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому мы должны понять их, прежде чем углубляться куда-либо ещё.

Объект может быть создан с помощью фигурных скобок `{…}` с необязательным списком *свойств*. Свойство – это пара «ключ: значение», где `ключ` – это строка (также называемая «именем свойства»), а `значение` может быть чем угодно.

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо.



Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

```
let user = new Object(); // синтаксис "конструктор объекта"
let user = {};  // синтаксис "литерал объекта"
```

Обычно используют вариант с фигурными скобками `{...}`. Такое объявление называют *литералом объекта* или *литеральной нотацией*.

Литералы и свойства

При использовании литерального синтаксиса `{...}` мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

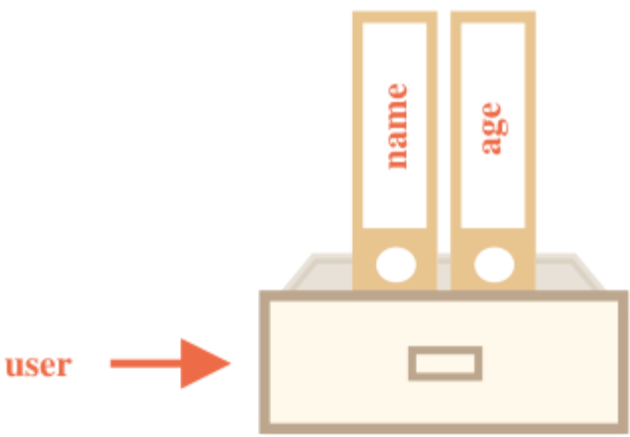
```
let user = {      // объект
  name: "John",   // под ключом "name" хранится значение "John"
  age: 30         // под ключом "age" хранится значение 30
};
```

У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие `":"`, и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте `user` сейчас находятся два свойства:

1. Первое свойство с именем `"name"` и значением `"John"`.
2. Второе свойство с именем `"age"` и значением `30`.

Можно сказать, что наш объект `user` – это ящик с двумя папками, подписанными «name» и «age».



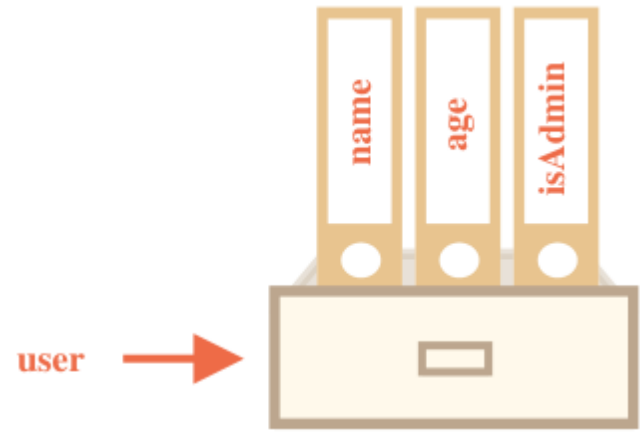
Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

```
// получаем свойства объекта:
alert( user.name ); // John
alert( user.age ); // 30
```

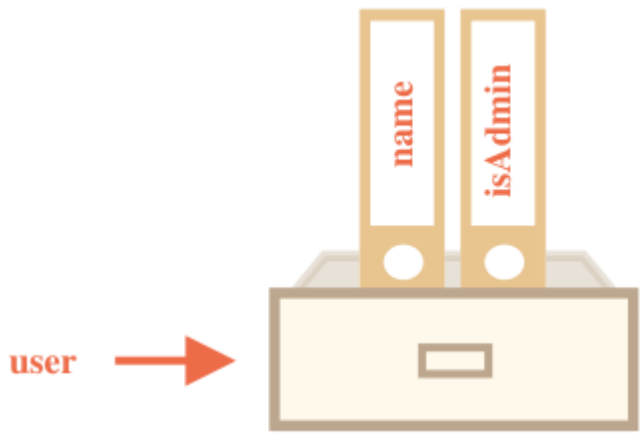
Значение может быть любого типа. Давайте добавим свойство с логическим значением:

```
user.isAdmin = true;
```



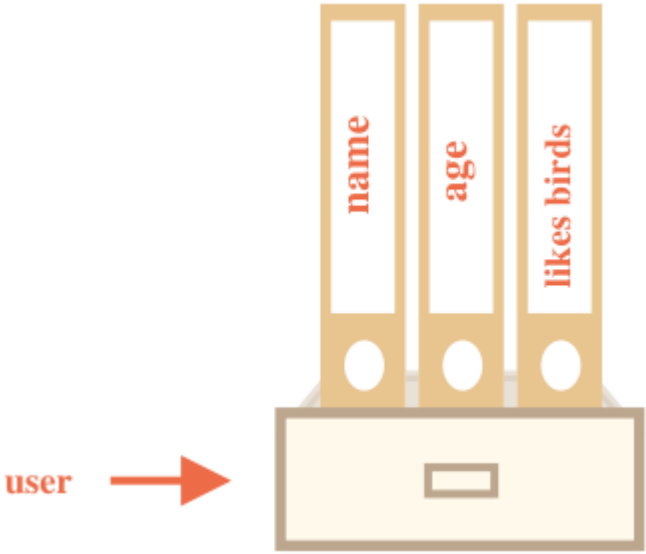
Для удаления свойства мы можем использовать оператор `delete`:

```
delete user.age;
```



Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

```
let user = {
  name: "John",
  age: 30,
  "likes birds": true  // имя свойства из нескольких слов должно быть в кавычках
};
```



Последнее свойство объекта может заканчиваться запятой:

```
let user = {
  name: "John",
  age: 30
}
,
```

Это называется «висячая запятая». Такой подход упрощает добавление, удаление и перемещение свойств, так как все строки объекта становятся одинаковыми.

Объект, объявленный как константа, может быть изменён

Объект, объявленный через `const`, *может* быть изменён.

Например:

```
const user = {
  name: "John"
};


alert(user.name); // Pete
user.name = "Pete"; // (*)
```

Может показаться, что строка `(*)` должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление `const` защищает от изменений только саму переменную `user`, а не её содержимое.

Определение `const` выдаст ошибку только если мы присвоим переменной другое значение: `user=...`.

Есть ещё один способ сделать константами свойства объекта, который мы рассмотрим в главе Флаги и дескрипторы свойств.

Квадратные скобки

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

```
// это вызовет синтаксическую ошибку
user.likes birds = true
```

JavaScript видит, что мы обращаемся к свойству `user.likes`, а затем идёт непонятное слово `birds`. В итоге синтаксическая ошибка.

Точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме `$` и `_`.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

```
let user = {};

// присваивание значения свойству
user["likes birds"] = true;

// получение значения свойства
alert(user["likes birds"]); // true

// удаление свойства
delete user["likes birds"];
```

Сейчас всё в порядке. Обратите внимание, что строка в квадратных скобках заключена в кавычки (подойдёт любой тип кавычек).

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

```
let key = "likes birds";

// то же самое, что и user["likes birds"] = true;
user[key] = true;
```

Здесь переменная `key` может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого мы используем её для доступа к

свойству. Это даёт нам большую гибкость.

Пример:

```
let user = {
  name: "John",
  age: 30
};

let key = prompt("Что вы хотите узнать о пользователе?",
"name");

// доступ к свойству через переменную
alert( user[key] ); // John (если ввели "name")
```

Запись «через точку» такого не позволяет:

```
let user = {
  name: "John",
  age: 30
};

let key = "name";
alert( user.key ); // undefined
```

Вычисляемые свойства

Мы можем использовать квадратные скобки в литеральной нотации для создания *вычисляемого свойства*.

Пример:

```
let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

};

alert( bag.apple ); // 5, если fruit="apple"
  [fruit]: 5, // имя свойства будет взято из
переменной fruit
```

Смысл вычисляемого свойства прост: запись `[fruit]` означает, что имя свойства необходимо взять из

переменной `fruit`.

И если посетитель введёт слово `"apple"`, то в объекте `bag` теперь будет лежать свойство `{apple: 5}`.

По сути, пример выше работает так же, как и следующий пример:

```
let fruit = prompt("Какой фрукт купить?", "apple");
let bag = {};

// имя свойства будет взято из переменной fruit
bag[fruit] = 5;
```

…Но первый пример выглядит лаконичнее.

Мы можем использовать и более сложные выражения в квадратных скобках:

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Квадратные скобки дают намного больше возможностей, чем запись через точку. Они позволяют использовать любые имена свойств и переменные, хотя и требуют более громоздких конструкций кода.

Подведём итог: в большинстве случаев, когда имена свойств известны и просты, используется запись через точку. Если же нам нужно что-то более сложное, то мы используем квадратные скобки.

Свойство из переменной

В реальном коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем.

Например:

```
function makeUser(name, age) {
  return {
    name: name,
    age: age
    // ...другие свойства
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

В примере выше название свойств `name` и `age` совпадают с названиями переменных, которые мы подставляем в качестве значений этих свойств. Такой подход настолько распространён, что существуют специальные *короткие свойства* для упрощения этой записи.

Вместо `name:name` мы можем написать просто `name`:

```
function makeUser(name, age) {

}
  return {
    name, // то же самое, что и name: name
    age   // то же самое, что и age: age
    // ...
  };
```

Мы можем использовать как обычные свойства, так и короткие в одном и том же объекте:

```
let user = {
  name,  // тоже самое, что и name:name
  age: 30
};
```

Ограничения на имена свойств

Как мы уже знаем, имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д.

Но для свойств объекта такого ограничения нет:

```
// эти имена свойств допустимы
let obj = {
  for: 1,
  let: 2,
  return: 3
};

alert( obj.for + obj.let + obj.return );  // 6
```

Иными словами, нет никаких ограничений к именам свойств. Они могут быть в виде строк или символов (специальный тип для идентификаторов, который будет рассмотрен позже).

Все другие типы данных будут автоматически преобразованы к строке.

Например, если использовать число `0` в качестве ключа, то оно превратится в строку `"0"`:

```
let obj = {
  0: "Тест" // то же самое что и "0": "Тест"
};

// обе функции alert выведут одно и то же свойство (число 0
преобразуется в строку "0")
alert( obj["0"] ); // Тест
alert( obj[0] ); // Тест (то же свойство)
```

Есть небольшой подводный камень, связанный со специальным свойством `__proto__`. Мы не можем установить его в необъектное значение:

```
let obj = {};
obj.__proto__ = 5; // присвоим число
alert(obj.__proto__); // [object Object], значение - это
```

```
объект, т.е. не то, что мы ожидали
```

Как мы видим, присвоение примитивного значения `5` игнорируется.

Мы более подробно исследуем особенности свойства `__proto__` в следующих главах Прототипное наследование, а также предложим способы исправления такого поведения.

Проверка существования свойства, оператор «in»

В отличие от многих других языков, особенность JavaScript-объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет!

При обращении к свойству, которого нет, возвращается `undefined`. Это позволяет просто проверить существование свойства:

```
let user = {};

alert( user.noSuchProperty === undefined ); // true означает
"свойства нет"
```

Также существует специальный оператор `"in"` для проверки существования свойства в объекте.

Синтаксис оператора:

```
"key" in object
```

Пример:

```
let user = { name: "John", age: 30 };
```

```
alert( "age" in user ); // true, user.age существует
alert( "blabla" in user ); // false, user.blabla не существует
```

Обратите внимание, что слева от оператора `in` должно быть *имя свойства*. Обычно это строка в кавычках.

Если мы опускаем кавычки, это значит, что мы указываем переменную, в которой находится имя свойства. Например:

```
let user = { age: 30 };

let key = "age";
alert(  in user ); // true, имя свойства было взято из переменной key
key
```

Для чего вообще нужен оператор `in`? Разве недостаточно сравнения с `undefined`?

В большинстве случаев прекрасно сработает сравнение с `undefined`. Но есть особый случай, когда оно не подходит, и нужно использовать `"in"`.

Это когда свойство существует, но содержит значение `undefined`:

```
let obj = {
  test: undefined
};

alert( obj.test ); //  выведет undefined, значит свойство не существует?
alert( "test" in obj ); // true, свойство существует!
```

В примере выше свойство `obj.test` технически существует в объекте. Оператор `in` сработал правильно.

Подобные ситуации случаются очень редко, так как `undefined` обычно явно не присваивается. Для

«неизвестных» или «пустых» свойств мы используем значение `null`. Таким образом, оператор `in` является экзотическим гостем в коде.

Цикл «for…in»

Для перебора всех свойств объекта используется цикл `for..in`. Этот цикл отличается от изученного ранее цикла `for(;;)`.

Синтаксис:

```
for (key in object) {
  // тело цикла выполняется для каждого свойства объекта
}
```

К примеру, давайте выведем все свойства объекта `user`:

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for (let key in user) {
  // ключи
  alert( key );  // name, age, isAdmin
  // значения ключей
  alert( user[key] ); // John, 30, true
}
```

Обратите внимание, что все конструкции «for» позволяют нам объявлять переменную внутри цикла, как, например, `let key` здесь.

Кроме того, мы могли бы использовать другое имя переменной. Например, часто используется вариант `"for (let prop in obj)"`.

Упорядочение свойств объекта

Упорядочены ли свойства объекта? Другими словами, если мы будем в цикле перебирать все свойства объекта, получим ли мы их в том же порядке, в котором мы их добавляли? Можем ли мы на это рассчитывать?

Короткий ответ: свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. Разберёмся подробнее.

В качестве примера рассмотрим объект с телефонными кодами:

```
let codes = {
  "49": "Германия",
  "41": "Швейцария",
  "44": "Великобритания",
  // ..,
  "1": "США"
};

for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

Если мы делаем сайт для немецкой аудитории, то, вероятно, мы хотим, чтобы код `49` был первым.

Но если мы запустим код, мы увидим совершенно другую картину:

США (1) идёт первым

затем Швейцария (41) и так далее.

Телефонные коды идут в порядке возрастания, потому что они являются целыми числами: `1, 41, 44, 49`.

Целочисленные свойства? Это что?

Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений.

То есть, `"49"` – это целочисленное имя свойства, потому что если его преобразовать в целое число, а затем обратно в строку, то оно не изменится. А вот свойства `"+49"` или `"1.2"` таковыми не являются:

```
// Math.trunc – встроенная функция, которая удаляет десятичную часть
alert( String(Math.trunc(Number("49"))) ); // "49", то же самое ⇒ свойство целочисленное
alert( String(Math.trunc(Number("+49"))) ); // "49", не то же самое, что "+49" ⇒ свойство не целочисленное
alert( String(Math.trunc(Number("1.2"))) ); // "1", не то же самое, что "1.2" ⇒ свойство не целочисленное
```

…С другой стороны, если ключи не целочисленные, то они перебираются в порядке создания, например:

```
let user = {
  name: "John",
  surname: "Smith"
};
user.age = 25; // добавим ещё одно свойство

for (let prop in user) {
  alert( prop ); // name, surname, age
}
// не целочисленные свойства перечислены в порядке создания
```

Таким образом, чтобы решить нашу проблему с телефонными кодами, мы можем схитрить, сделав коды не целочисленными свойствами. Добавления знака `"+"` перед каждым кодом будет достаточно.

Пример:

```
let codes = {
  "+49": "Германия",
  "+41": "Швейцария",
  "+44": "Великобритания",
  // ..,
  "+1": "США"
};

for (let code in codes) {
  alert( +code ); // 49, 41, 44, 1
}
```

Теперь код работает так, как мы задумывали.

Итого

Объекты – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

- Ключи свойств должны быть строками или символами (обычно строками).
- Значения могут быть любого типа.

Чтобы получить доступ к свойству, мы можем использовать:

- Запись через точку: `obj.property`.
- Квадратные скобки `obj["property"]`. Квадратные скобки позволяют взять ключ из переменной, например, `obj[varWithKey]`.

Дополнительные операторы:

- Удаление свойства: `delete obj.prop`.

Проверка существования свойства: `"key" in obj`.

Перебор свойств объекта: цикл for `for (let key in obj)`.

То, что мы изучали в этой главе, называется «простым объектом» («plain object») или просто `Object`.

В JavaScript есть много других типов объектов:

`Array` для хранения упорядоченных коллекций данных,

`Date` для хранения информации о дате и времени,

`Error` для хранения информации об ошибке.

… и так далее.

У них есть свои особенности, которые мы изучим позже. Иногда люди говорят что-то вроде «тип данных Array» или «тип данных Date», но формально они не являются отдельными типами, а относятся к типу данных `Object`. Они лишь расширяют его различными способами.

Объекты в JavaScript очень мощные. Здесь мы только немного углубились в действительно огромную тему. Мы будем плотно работать с объектами и узнаем о них больше в следующих частях учебника.

Задачи

Привет, object

важность: 5

Напишите код, выполнив задание из каждого пункта отдельной строкой:

1. Создайте пустой объект `user`.
2. Добавьте свойство `name` со значением `John`.
3. Добавьте свойство `surname` со значением `Smith`.
4. Измените значение свойства `name` на `Pete`.
5. Удалите свойство `name` из объекта.

К решению

Проверка на пустоту

важность: 5

Напишите функцию `isEmpty(obj)`, которая возвращает `true`, если у объекта нет свойств, иначе `false`.

Должно работать так:

```
let schedule = {};

alert( isEmpty(schedule) ); // true

schedule["8:30"] = "get up";

alert( isEmpty(schedule) ); // false
```

Открыть песочницу с тестами для задачи.

К решению

Объекты-константы?

важность: 5

Можно ли изменить объект, объявленный с помощью `const`? Как вы думаете?

```
const user = {
  name: "John"
};

// это будет работать?
user.name = "Pete";
```

К решению

Сумма свойств объекта

важность: 5

У нас есть объект, в котором хранятся зарплаты нашей команды:

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
}
```

Напишите код для суммирования всех зарплат и сохраните результат в переменной `sum`. Должно получиться `390`.

Если объект `salaries` пуст, то результат должен быть `0`.

К решению

Умножаем все числовые свойства на 2

важность: 3

Создайте функцию `multiplyNumeric(obj)`, которая умножает все числовые свойства объекта `obj` на `2`.

Например:

```
// до вызова функции
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};

multiplyNumeric(menu);

// после вызова функции
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

Обратите внимание, что `multiplyNumeric` не нужно ничего возвращать. Следует напрямую изменять объект.

P.S. Используйте `typeof` для проверки, что значение свойства числовое.

Открыть песочницу с тестами для задачи.

К решению

Копирование объектов и ссылки

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке».

Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению».

Например:

```
let message = "Привет!";
let phrase = message;
```

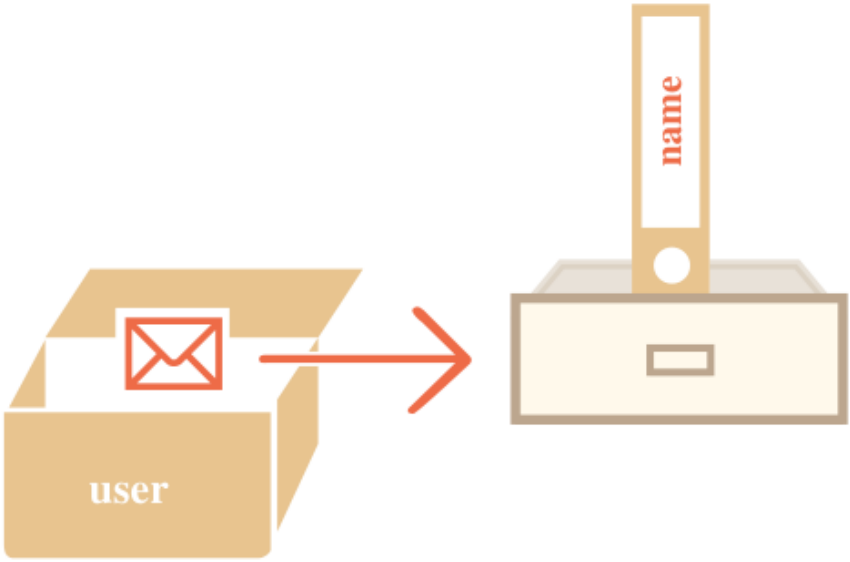
В результате мы имеем две независимые переменные, каждая из которых хранит строку `"Привет!"`.

Объекты ведут себя иначе.

Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него.

Проиллюстрируем это:

```
let user = {
  name: "Иван"
};
```



Сам объект хранится где-то в памяти. А в переменной `user` лежит «ссылка» на эту область памяти.

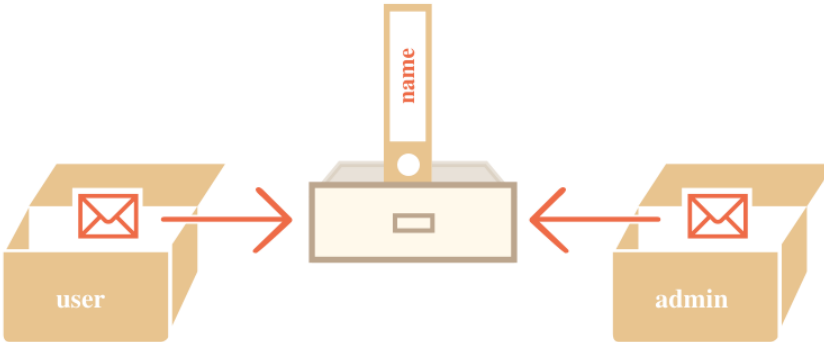
Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется.

Если мы представляем объект как ящик, то переменная – это ключ к нему. Копирование переменной дублирует ключ, но не сам ящик.

Например:

```
let user = { name: "Иван" };

let admin = user; // копируется ссылка
```

Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект:



Мы можем использовать любую из переменных для доступа к ящику и изменения его содержимого:

```
let user = { name: 'Иван' };

let admin = user;


alert(); // 'Петя', изменения видны по ссылке из переменной "user"
admin.name = 'Петя'; // изменено по ссылке из переменной "admin"
user.name
```

Приведённый выше пример демонстрирует, что объект только один. Как если бы у нас был один ящик с двумя ключами и мы использовали один из них (`admin`), чтобы войти в него и что-то изменить, а затем, открыв ящик другим ключом (`user`), мы бы увидели эти изменения.

Сравнение по ссылке

Операторы равенства `==` и строгого равенства `===` для объектов работают одинаково.

Два объекта равны только в том случае, если это один и тот же объект.

В примере ниже две переменные ссылаются на один и тот же объект, поэтому они равны друг другу:

```
let a = {};
let b = a; // копирование по ссылке

alert( a == b ); // true, т.к. обе переменные ссылаются на
один и тот же объект
alert( a === b ); // true
```

В другом примере два разных объекта не равны, хотя оба пусты:

```
let a = {};
let b = {}; // два независимых объекта

alert( a == b ); // false
```

Для сравнений типа `obj1 > obj2` или для сравнения с примитивом `obj == 5` объекты преобразуются в примитивы. Мы скоро изучим, как работают такие преобразования объектов, но, по правде говоря, сравнения такого рода необходимы очень редко и обычно являются результатом ошибки программиста.

Клонирование и объединение объектов, Object.assign

Таким образом, при копировании переменной с объектом создаётся ещё одна ссылка на тот же самый объект.

Но что, если нам всё же нужно дублировать объект? Создать независимую копию, клон?

Это выполнимо, но немного сложно, так как в JavaScript нет встроенного метода для этого. На самом деле, такая нужда возникает редко. В большинстве случаев нам достаточно копирования по ссылке.

Но если мы действительно этого хотим, то нам нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их.

Например так:

```
let user = {
  name: "Иван",
  age: 30
};



// теперь в переменной clone находится абсолютно независимый
клон объекта
clone.name = "Пётр"; // изменим в нём данные

alert( user.name ); // в оригинальном объекте значение
свойства `name` осталось прежним – Иван.
let clone = {}; // новый пустой объект

// скопируем все свойства user в него
for (let key in user) {
  clone[key] = user[key];
}
```

Кроме того, для этих целей мы можем использовать метод Object.assign.

Синтаксис:

```
Object.assign(dest, [src1, src2, src3...])
```

Первый аргумент `dest` — целевой объект.

Остальные аргументы `src1, ..., srcN` (может быть столько, сколько нужно)) являются исходными объектами

Метод копирует свойства всех исходных объектов `src1, ..., srcN` в целевой объект `dest`. То есть, свойства всех перечисленных объектов, начиная со

второго, копируются в первый объект.

Возвращает объект `dest`.

Например, объединим несколько объектов в один:

```
let user = { name: "Иван" };

let permissions1 = { canView: true };
let permissions2 = { canEdit: true };



// теперь user = { name: "Иван", canView: true, canEdit: true
}
// копируем все свойства из permissions1 и
permissions2 в user
Object.assign(user, permissions1,
permissions2);
```

Если принимающий объект (`user`) уже имеет свойство с таким именем, оно будет перезаписано:

```
let user = { name: "Иван" };

Object.assign(user, { name: "Пётр" });

alert(user.name); // теперь user = { name: "Пётр" }
```

Мы также можем использовать `Object.assign` для замены `for..in` на простое клонирование:

```
let user = {
  name: "Иван",
  age: 30
};

let clone = Object.assign({}, user);
```

Этот метод скопирует все свойства объекта `user` в пустой объект и возвратит его.

Вложенное клонирование

До сих пор мы предполагали, что все свойства объекта `user` хранят примитивные значения. Но свойства могут быть ссылками на другие объекты. Что с ними делать?

Например, есть объект:

```
let user = {
  name: "Иван",
  sizes: {
    height: 182,
    width: 50
  }
};

alert( user.sizes.height ); // 182
```

Теперь при клонировании недостаточно просто скопировать `clone.sizes = user.sizes`, поскольку `user.sizes` – это объект, он будет скопирован по ссылке. А значит объекты `clone` и `user` в своих свойствах `sizes` будут ссылаться на один и тот же объект:

```
let user = {
  name: "Иван",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, один и тот же объект

// user и clone обращаются к одному sizes
user.sizes.width++;       // меняем свойство в одном объекте
alert(clone.sizes.width); // 51, видим результат в другом объекте
```

Чтобы исправить это, мы должны в цикле клонирования

делать проверку, не является ли значение `user[key]` объектом, и если это так – скопировать и его структуру тоже. Это называется «глубокое клонирование».

Мы можем реализовать глубокое клонирование, используя рекурсию. Или, чтобы не изобретать велосипед, использовать готовую реализацию — метод _.cloneDeep(obj) из JavaScript-библиотеки lodash.

Итого

Объекты присваиваются и копируются по ссылке. Другими словами, переменная хранит не «значение объекта», а «ссылку» (адрес в памяти) на это значение. Поэтому копирование такой переменной или передача её в качестве аргумента функции приводит к копированию этой ссылки, а не самого объекта.

Все операции с использованием скопированных ссылок (например, добавление или удаление свойств) выполняются с одним и тем же объектом.

Для «простого клонирования» объекта можно использовать `Object.assign`. Необходимо помнить, что `Object.assign` не делает глубокое клонирования объекта. Если внутри копируемого объекта есть свойство значение, которого не является примитивом, оно будет передано по ссылке. Для создания «настоящей копии» (полного клона объекта) можно воспользоваться методом из сторонней JavaScript-библиотеки _.cloneDeep(obj).

Сборка мусора

Управление памятью в JavaScript выполняется автоматически и незаметно. Мы создаём примитивы, объекты, функции… Всё это занимает память.

Но что происходит, когда что-то больше не нужно? Как JavaScript понимает, что пора очищать память?

Достижимость

Основной концепцией управления памятью в JavaScript является принцип *достижимости*.

Если упростить, то «достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

1. Существует базовое множество достижимых значений, которые не могут быть удалены.

   Например:

   Локальные переменные и параметры текущей функции.

   Переменные и параметры других функций в текущей цепочке вложенных вызовов.

   Глобальные переменные.

   (некоторые другие внутренние значения)

   Эти значения мы будем называть *корнями*.

6. Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

   Например, если в локальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы. Далее вы познакомитесь с подробными примерами на эту тему.

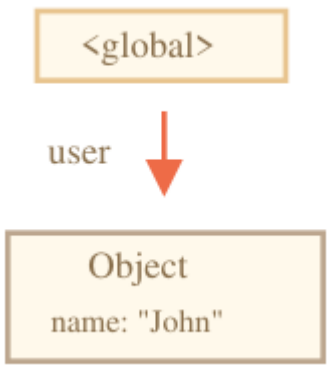
В интерпретаторе JavaScript есть фоновый процесс,

который называется сборщик мусора. Он следит за всеми объектами и удаляет те, которые стали недостижимы.

Простой пример

Вот самый простой пример:

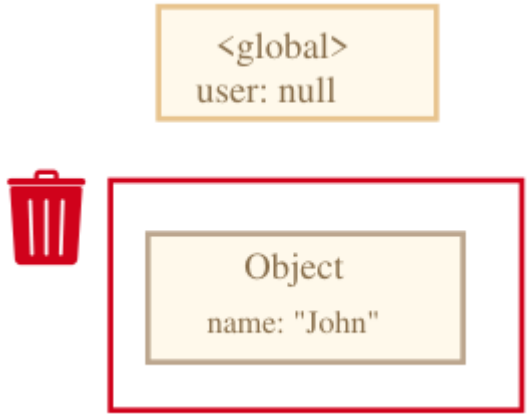
```
// в user находится ссылка на объект
let user = {
  name: "John"
};
```



Здесь стрелка обозначает ссылку на объект. Глобальная переменная `user` ссылается на объект `{name: "John"}` (мы будем называть его просто «John»). В свойстве `"name"` объекта John хранится примитив, поэтому оно нарисовано внутри объекта.

Если перезаписать значение `user`, то ссылка потеряется:

```
user = null;
```

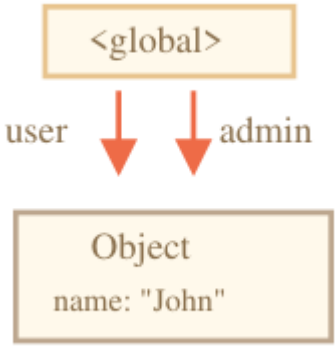


Теперь объект John становится недостижимым. К нему нет доступа, на него нет ссылок. Сборщик мусора удалит эти данные и освободит память.

Две ссылки

Представим, что мы скопировали ссылку из `user` в `admin`:

```
// в user находится ссылка на объект
let user = {
  name: "John"
};

let admin = user;
```



Теперь, если мы сделаем то же самое:

```
user = null;
```

…то объект John всё ещё достижим через глобальную переменную `admin`, поэтому он находится в памяти. Если бы мы также перезаписали `admin`, то John был бы удалён.

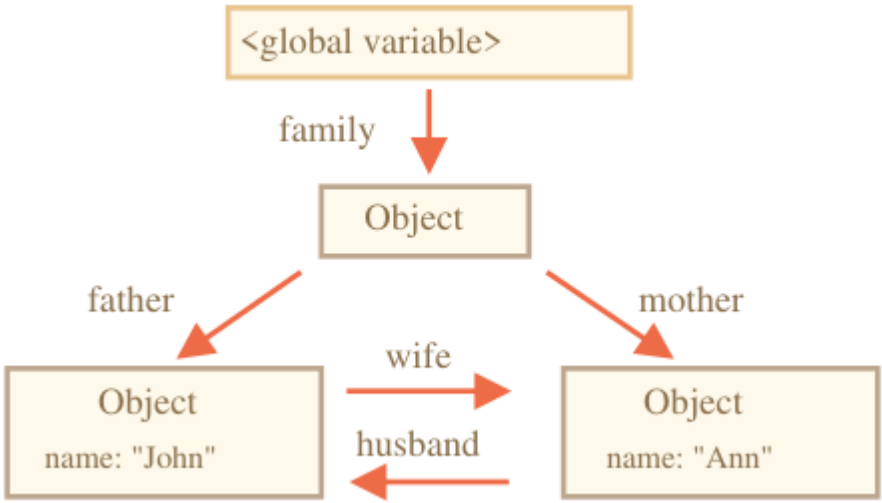
Взаимосвязанные объекты

Теперь более сложный пример. Семья:

```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

  return {
    father: man,
    mother: woman
  }
}

let family = marry({
  name: "John"
}, {
  name: "Ann"
});
```

Функция `marry` «женит» два объекта, давая им ссылки друг на друга, и возвращает новый объект, содержащий ссылки на два предыдущих.

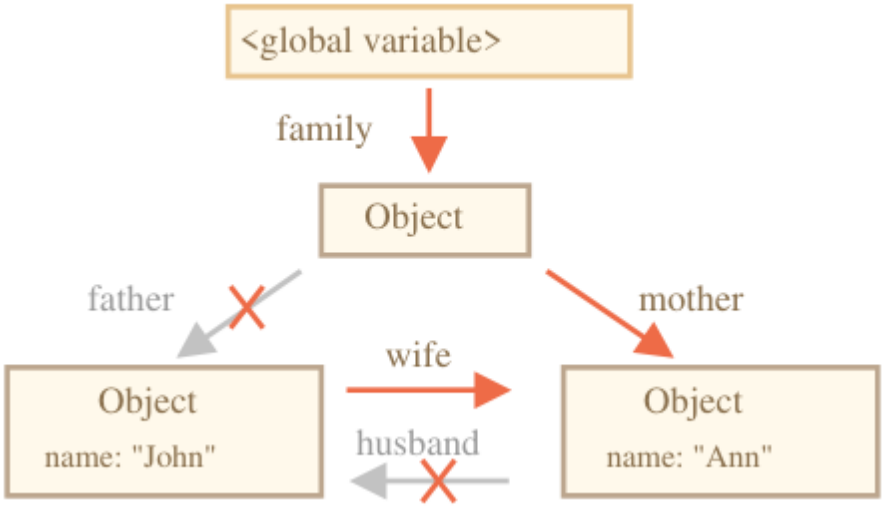
В результате получаем такую структуру памяти:



На данный момент все объекты достижимы.

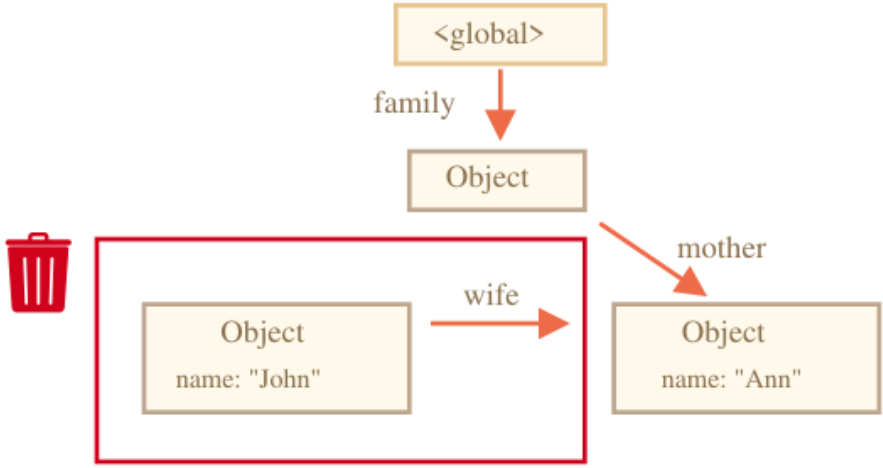
Теперь удалим две ссылки:

```
delete family.father;
delete family.mother.husband;
```



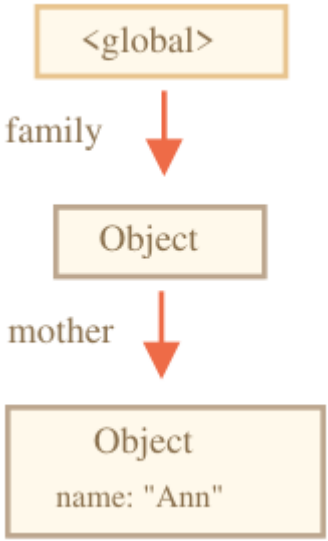
Недостаточно удалить только одну из этих ссылок, потому что все объекты останутся достижимыми.

Но если мы удалим обе, то увидим, что у объекта John больше нет входящих ссылок:



Исходящие ссылки не имеют значения. Только входящие ссылки могут сделать объект достижимым. Объект John теперь недостижим и будет удалён из памяти со всеми своими данными, которые также стали недоступны.

После сборки мусора:



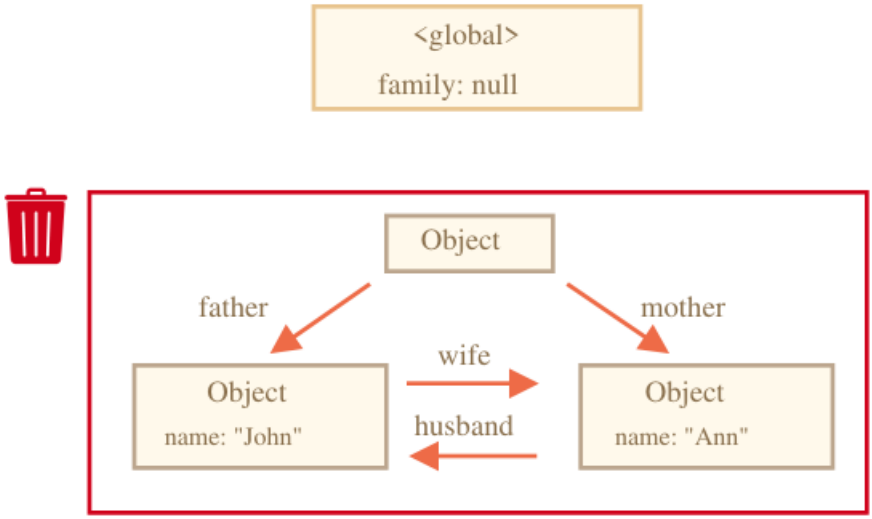
Недостижимый «остров»

Вполне возможна ситуация, при которой целый «остров» связанных объектов может стать недостижимым и удалиться из памяти.

Возьмём объект `family` из примера выше. А затем:

```
family = null;
```

Структура в памяти теперь станет такой:



Этот пример демонстрирует, насколько важна концепция достижимости.

Объекты John и Ann всё ещё связаны, оба имеют входящие ссылки, но этого недостаточно.

У объекта `family` больше нет ссылки от корня, поэтому весь «остров» становится недостижимым и будет удалён.

Внутренние алгоритмы

Основной алгоритм сборки мусора – «алгоритм пометок» (англ. «mark-and-sweep»).

Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

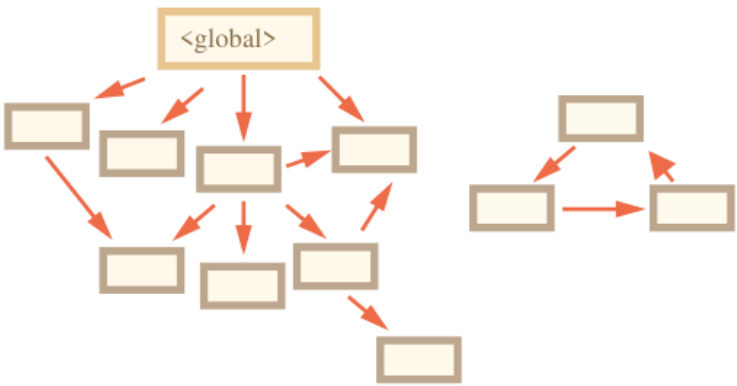
- Сборщик мусора «помечает» (запоминает) все корневые объекты.
- Затем он идёт по их ссылкам и помечает все найденные объекты.
- Затем он идёт по ссылкам помеченных объектов и

помечает объекты, на которые есть ссылка от них. Все объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.

…И так далее, пока не будут посещены все ссылки (достижимые от корней).

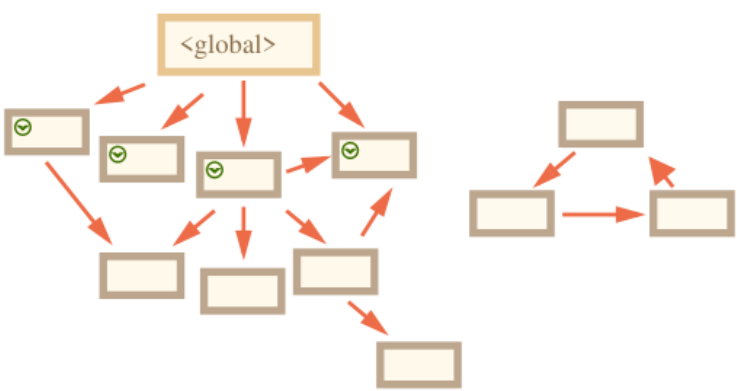
Все непомеченные объекты удаляются.

Например, пусть наша структура объектов выглядит так:

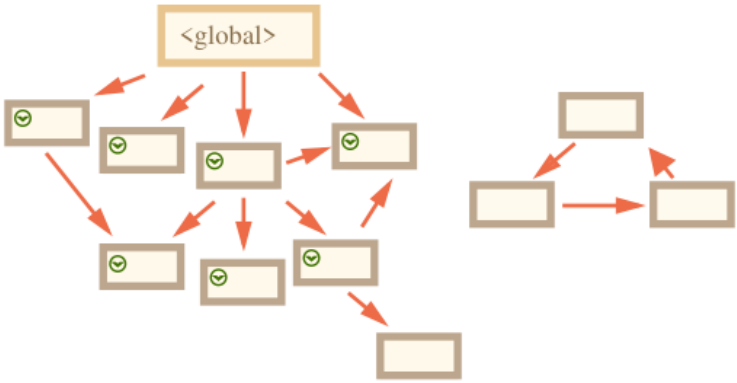


Явно виден «недостижимый остров» справа. Теперь посмотрим, как будет работать «алгоритм пометок» сборщика мусора.

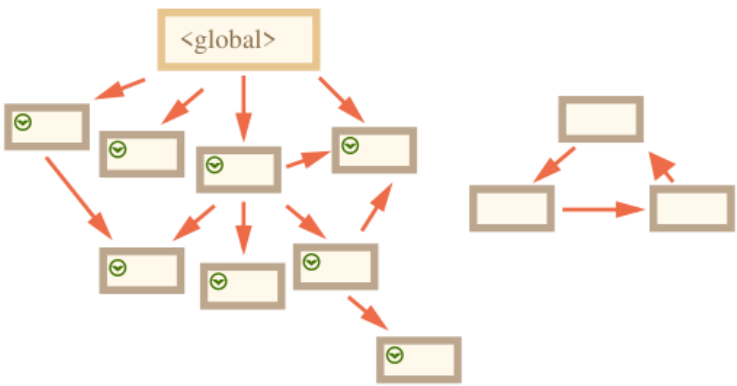
На первом шаге помечаются корни:



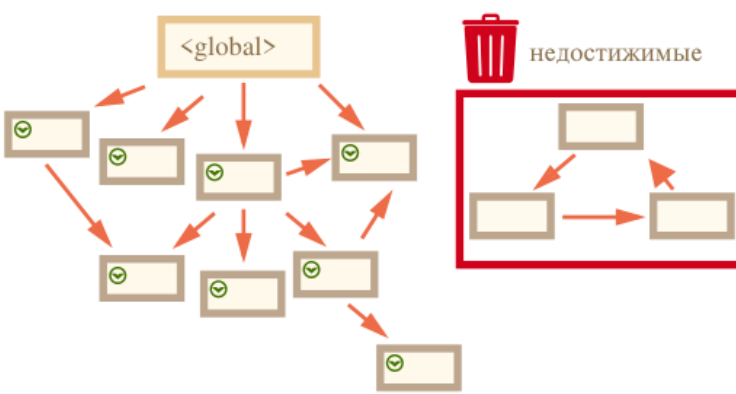
Затем помечаются объекты по их ссылкам:



…а затем объекты по их ссылкам и так далее, пока это вообще возможно:



Теперь объекты, до которых не удалось дойти от корней, считаются недостижимыми и будут удалены:



Это и есть принцип работы сборки мусора.

Интерпретаторы JavaScript применяют множество оптимизаций, чтобы сборка мусора работала быстрее и не влияла на производительность.

Вот некоторые из оптимизаций:

Сборка по поколениям (Generational collection) – объекты делятся на «новые» и «старые». Многие объекты появляются, выполняют свою задачу и быстро умирают, их можно удалять более агрессивно. Те, которые живут достаточно долго, становятся «старыми» и проверяются реже.

Инкрементальная сборка (Incremental collection) – если объектов много, то обход всех ссылок и пометка достижимых объектов может занять значительное время и привести к видимым задержкам выполнения скрипта. Поэтому интерпретатор пытается организовать сборку мусора поэтапно. Этапы выполняются по отдельности один за другим. Это требует дополнительного учёта для отслеживания изменений между этапами, но зато теперь у нас есть много крошечных задержек вместо одной большой.

Сборка в свободное время (Idle-time collection) – чтобы уменьшить возможное влияние на производительность, сборщик мусора старается работать только во время простоя процессора.

Существуют и другие способы оптимизации и разновидности алгоритмов сборки мусора. Но как бы мне ни хотелось описать их здесь, я должен воздержаться от этого, потому что разные интерпретаторы JavaScript применяют разные приёмы и хитрости. И, что более важно, всё меняется по мере развития интерпретаторов, поэтому углубляться в эту тему заранее, без реальной необходимости, вероятно, не стоит. Если, конечно, это не вопрос чистого интереса, тогда для вас будут полезны некоторые ссылки ниже.

Итого

Главное из того, что мы узнали:

> Сборка мусора выполняется автоматически. Мы не можем ускорить или предотвратить её.
>
> Объекты сохраняются в памяти, пока они достижимы.
>
> Наличие ссылки не гарантирует, что объект достижим (от корня): несколько взаимосвязанных объектов могут стать недостижимыми как единое целое.

Современные интерпретаторы реализуют передовые алгоритмы сборки мусора.

Некоторые из них освещены в книге «The Garbage Collection Handbook: The Art of Automatic Memory Management» (R. Jones и др.).

Если вы знакомы с низкоуровневым программированием, то более подробная информация о сборщике мусора интерпретатора V8 находится в статье A tour of V8: Garbage Collection.

Также в блоге интерпретатора V8 время от времени публикуются статьи об изменениях в управлении памятью. Разумеется, чтобы изучить сборку мусора, вам необходимо понимать, как устроен внутри интерпретатор V8 в целом. Об этом вы можете почитать в блоге Вячеслава Егорова, одного из инженеров, разрабатывавших V8. Я говорю про «V8», потому что он лучше всего освещён статьями в интернете. В других интерпретаторах многие подходы схожи, но сборка мусора во многих аспектах отличается.

Глубокое понимание работы интерпретаторов необходимо, когда вам нужны низкоуровневые оптимизации. Было бы разумно запланировать их изучение как следующий шаг после освоения языка.

Методы объекта, "this"

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

```
// Объект пользователя
let user = {
  name: "Джон",
  age: 30
};
```

И так же, как и в реальном мире, пользователь может *совершать действия*: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены свойствами-функциями объекта.

Примеры методов

Для начала давайте научим нашего пользователя `user` здороваться:

```
let user = {
  name: "Джон",
  age: 30
};
```

```
user.sayHi(); // Привет!
user.sayHi = function() {
  alert("Привет!");
};
```

Здесь мы просто использовали Function Expression (функциональное выражение), чтобы создать функцию для приветствия, и присвоили её свойству `user.sayHi`

нашего объекта.

Затем мы вызвали её. Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют *методом* этого объекта.

Итак, мы получили метод `sayHi` объекта `user`.

Конечно, мы могли бы заранее объявить функцию и использовать её в качестве метода, примерно так:

```
let user = {
  // ...
};


user.sayHi(); // Привет!
// сначала объявляем
function sayHi() {
  alert("Привет!");
};

// затем добавляем в качестве метода
user.sayHi = sayHi;
```

Объектно-ориентированное программирование

Когда мы пишем наш код, используя объекты для представления сущностей реального мира, – это называется объектно-ориентированное программирование или сокращённо: «ООП».

ООП является большой предметной областью и интересной наукой само по себе. Как выбрать правильные сущности? Как организовать

взаимодействие между ними? Это – создание архитектуры, и есть хорошие книги по этой теме, такие как «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» авторов Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес или «Объектно-ориентированный анализ и проектирование с примерами приложений» Гради Буча, а также ещё множество других книг.

Сокращённая запись метода

Существует более короткий синтаксис для методов в литерале объекта:

```
// эти объекты делают одно и то же (одинаковые методы)

user = {
  sayHi: function() {
    alert("Привет");
  }
};

// сокращённая запись выглядит лучше, не так ли?
user = {

    alert("Привет");
  }
};
  sayHi() { // то же самое, что и "sayHi: function()"
```

Как было показано, мы можем пропустить ключевое слово `"function"` и просто написать `sayHi()`.

Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов (что будет рассмотрено позже), но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

Ключевое слово «this» в методах

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри `user.sayHi()` может понадобиться имя пользователя, которое хранится в объекте `user`.

Для доступа к информации внутри объекта метод может использовать ключевое слово `this`.

Значение `this` – это объект «перед точкой», который использовался для вызова метода.

Например:

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {

  }

};

user.sayHi(); // Джон
     // this - это "текущий объект"
     alert(this.name);
```

Здесь во время выполнения кода `user.sayHi()` значением `this` будет являться `user` (ссылка на объект `user`).

Технически также возможно получить доступ к объекту без ключевого слова `this`, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {

  }

};
    alert(user.name); // используем переменную
"user" вместо ключевого слова "this"
```

…Но такой код будет ненадёжным. Если мы решим скопировать ссылку на объект `user` в другую переменную, например, `admin = user`, и перезапишем переменную `user` чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из `admin`.

Это показано ниже:

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {

  }

};


let admin = user;
user = null; // обнулим переменную для наглядности, теперь она
не хранит ссылку на объект.

admin.sayHi(); // Ошибка! Внутри sayHi() используется user,
которая больше не ссылается на объект!
    alert( user.name ); // приведёт к ошибке
```

Если мы используем `this.name` вместо `user.name` внутри `alert`, тогда этот код будет работать.

«this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции.

В этом коде нет синтаксической ошибки:

```
function sayHi() {
  alert( .name );
}
this
```

Значение `this` вычисляется во время выполнения кода и зависит от контекста.

Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

```
let user = { name: "Джон" };
let admin = { name: "Админ" };

function sayHi() {
  alert( this.name );
}



// вызовы функции, приведённые ниже, имеют разное значение this
// "this" внутри функции является ссылкой на объект, который указан "перед точкой"
user.f(); // Джон  (this == user)
admin.f(); // Админ  (this == admin)

admin['f'](); // Админ (неважен способ доступа к методу - через точку или квадратные скобки)
// используем одну и ту же функцию в двух объектах
user.f = sayHi;
admin.f = sayHi;
```

Правило простое: при вызове `obj.f()` значение `this` внутри `f` равно `obj`. Так что, в приведённом примере это

`user` или `admin`.

Вызов без объекта: `this == undefined`

Мы даже можем вызвать функцию вовсе без использования объекта:

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

В строгом режиме (`"use strict"`) в таком коде значением `this` будет являться `undefined`. Если мы попытаемся получить доступ к `name`, используя `this.name` – это вызовет ошибку.

В нестрогом режиме значением `this` в таком случае будет *глобальный объект* (`window` для браузера, мы вернёмся к этому позже в главе Глобальный объект). Это – исторически сложившееся поведение `this`, которое исправляется использованием строгого режима (`"use strict"`).

Обычно подобный вызов является ошибкой программирования. Если внутри функции используется `this`, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

Последствия свободного `this`

Если вы до этого изучали другие языки программирования, тогда вы, скорее всего,

привыкли к идее "фиксированного `this`" – когда методы, определённые внутри объекта, всегда сохраняют в качестве значения `this` ссылку на свой объект (в котором был определён метод).

В JavaScript `this` является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Эта идея вычисления `this` в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Здесь мы не будем судить о том, является ли это решение в языке хорошим или плохим. Мы должны понимать, как с этим работать, чтобы получать выгоды и избегать проблем.

Внутренняя реализация: Ссылочный тип

Продвинутая возможность языка

Этот раздел объясняет сложную тему, чтобы лучше понимать некоторые запутанные случаи.

Если вы хотите продвигаться быстрее, его можно пропустить или отложить.

Некоторые хитрые способы вызова метода приводят к потере значения `this`, например:

```
let user = {
  name: "Джон",
  hi() { alert(this.name); },
  bye() { alert("Пока"); }
};

user.hi(); // Джон (простой вызов метода работает хорошо)

// теперь давайте попробуем вызывать user.hi или user.bye
// в зависимости от имени пользователя
user.name
(user.name == "Джон" ? user.hi : user.bye)();
// Ошибка!
```

В последней строчке кода используется условный оператор `?`, который определяет, какой будет вызван метод (`user.hi` или `user.bye`) в зависимости от выполнения условия. В данном случае будет выбран `user.hi`.

Затем метод тут же вызывается с помощью скобок `()`. Но вызов не работает как положено!

Вы можете видеть, что при вызове будет ошибка, потому что значением `"this"` внутри функции становится `undefined` (полагаем, что у нас строгий режим).

Так работает (доступ к методу объекта через точку):

```
user.hi();
```

Так уже не работает (вызываемый метод вычисляется):

```
(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!
```

Почему? Если мы хотим понять, почему так происходит, давайте разберёмся (заглянем под капот), как работает вызов методов (`obj.method()`).

Присмотревшись поближе, в выражении `obj.method()` можно заметить две операции:

1. Сначала оператор точка `'.'` возвращает свойство объекта – его метод (`obj.method`).
2. Затем скобки `()` вызывают этот метод (исполняется код метода).

Итак, каким же образом информация о `this` передаётся из первой части во вторую?

Если мы поместим эти операции в отдельные строки, то значение `this`, естественно, будет потеряно:

```
let user = {
  name: "Джон",
  hi() { alert(this.name); }
};

// разделим получение метода объекта и его
вызов в разных строках
let hi = user.hi;
hi(); // Ошибка, потому что значением this
является undefined
```

Здесь `hi = user.hi` сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет `this`.

Для работы вызовов типа `user.hi()`, JavaScript использует трюк – точка `'.'` возвращает не саму функцию, а специальное значение «ссылочного типа», называемого Reference Type.

Этот ссылочный тип (Reference Type) является внутренним типом. Мы не можем явно использовать его, но он используется внутри языка.

Значение ссылочного типа – это «триплет»: комбинация из трёх значений `(base, name, strict)`, где:

`base` – это объект.

`name` – это имя свойства объекта.

`strict` – это режим исполнения. Является true, если действует строгий режим (`use strict`).

Результатом доступа к свойству `user.hi` является не функция, а значение ссылочного типа. Для `user.hi` в строгом режиме оно будет таким:

```
// значение ссылочного типа (Reference Type)
(user, "hi", true)
```

Когда скобки `()` применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный `this` (=`user` в данном случае, по `base`).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки `.` до вызывающих скобок `()`.

При любой другой операции, например, присваивании `hi = user.hi`, ссылочный тип заменяется на собственно значение `user.hi` (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без `this`.

Таким образом, значение `this` передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки `obj.method()` или квадратных скобок `obj['method']()` (они делают то же самое). Позднее в этом учебнике мы изучим различные варианты решения проблемы потери значения `this`. Например, такие как func.bind().

У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» `this`. Если мы используем `this` внутри стрелочной функции, то его значение берётся из внешней «нормальной» функции.

Например, здесь `arrow()` использует значение `this` из внешнего метода `user.sayHi()`:

```
let user = {
  firstName: "Илья",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Илья
```

Это является особенностью стрелочных функций. Они полезны, когда мы на самом деле не хотим иметь отдельное значение `this`, а хотим брать его из внешнего контекста. Позднее в главе Повторяем стрелочные функции мы увидим больше примеров на эту тему.

Итого

> Функции, которые находятся в объекте в качестве его свойств, называются «методами».
>
> Методы позволяют объектам «действовать»: `object.doSomething()`.
>
> Методы могут ссылаться на объект через `this`.

Значение `this` определяется во время исполнения кода.

При объявлении любой функции в ней можно использовать `this`, но этот `this` не имеет значения до тех пор, пока функция не будет вызвана.

Эта функция может быть скопирована между объектами (из одного объекта в другой).

Когда функция вызывается синтаксисом «метода» – `object.method()`, значением `this` во время вызова является объект перед точкой.

Также ещё раз заметим, что стрелочные функции являются особенными – у них нет `this`. Когда внутри стрелочной функции обращаются к `this`, то его значение берётся снаружи.

Задачи

Проверка синтаксиса

важность: 2

Каким будет результат выполнения этого кода?

```
let user = {
  name: "Джон",
  go: function() { alert(this.name) }
}

(user.go)()
```

P.S. Здесь есть подвох :)

К решению

Объясните значение "this"

важность: 3

В представленном ниже коде мы намерены вызвать `obj.go()` метод 4 раза подряд.

Но вызовы `(1)` и `(2)` работают иначе, чем `(3)` и `(4)`. Почему?

```
let obj, method;

obj = {
  go: function() { alert(this); }
};

obj.go();               // (1) [object Object]

(obj.go)();             // (2) [object Object]

(method = obj.go)();    // (3) undefined

(obj.go || obj.stop)(); // (4) undefined
```

К решению

Использование "this" в литерале объекта

важность: 5

Здесь функция `makeUser` возвращает объект.

Каким будет результат при обращении к свойству объекта `ref`? Почему?

```
function makeUser() {
  return {
    name: "Джон",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // Каким будет результат?
```

К решению

Создайте калькулятор

важность: 5

Создайте объект `calculator` (калькулятор) с тремя методами:

- `read()` (читать) запрашивает два значения и сохраняет их как свойства объекта.
- `sum()` (суммировать) возвращает сумму сохранённых значений.
- `mul()` (умножить) перемножает сохранённые значения и возвращает результат.

```
let calculator = {
  // ... ваш код ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Запустить демо

Открыть песочницу с тестами для задачи.

К решению

Цепь вызовов

важность: 2

Это `ladder` (лестница) – объект, который позволяет подниматься вверх и спускаться:

```
let ladder = {
  step: 0,
  up() {
    this.step++;
  },
  down() {
    this.step--;
  },
  showStep: function() { // показывает текущую ступеньку
    alert( this.step );
  }
};
```

Теперь, если нам нужно сделать несколько последовательных вызовов, мы можем выполнить это так:

```
ladder.up();
ladder.up();
ladder.down();
ladder.showStep(); // 1
```

Измените код методов `up`, `down` и `showStep` таким образом, чтобы их вызов можно было сделать по цепочке, например так:

```
ladder.up().up().down().showStep(); // 1
```

Такой подход широко используется в библиотеках JavaScript.

Открыть песочницу с тестами для задачи.

К решению

Конструкторы, создание объектов через "new"

Обычный синтаксис `{...}` позволяет создать только один объект. Но зачастую нам нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д.

Это можно сделать при помощи функции-конструктора и оператора `"new"`.

Функция-конструктор

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора `"new"`.

Например:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}



alert(user.name); // Вася
alert(user.isAdmin); // false
let user = new User("Вася");
```

Когда функция вызывается как `new User(...)`, происходит следующее:

1. Создаётся новый пустой объект, и он присваивается `this`.
2. Выполняется код функции. Обычно он модифицирует `this`, добавляет туда новые свойства.

3. Возвращается значение `this`.

Другими словами, вызов `new User(...)` делает примерно вот что:

```
function User(name) {


  // добавляет свойства к this
  this.name = name;
  this.isAdmin = false;


}
   // this = {};   (неявно)
   // return this;   (неявно)
```

То есть, результат вызова `new User("Вася")` – это тот же объект, что и:

```
let user = {
  name: "Вася",
  isAdmin: false
};
```

Теперь, когда нам необходимо будет создать других пользователей, мы можем использовать `new User("Маша")`, `new User("Даша")` и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора `new`, и выполнится алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-

конструктором, и её нужно вызывать через `new`.

new function() { … }

Если в нашем коде большое количество строк, создающих один сложный объект, мы можем обернуть их в функцию-конструктор следующим образом:

```
let user = new function() {
  this.name = "Вася";
  this.isAdmin = false;

  // ...другой код для создания пользователя
  // возможна любая сложная логика и выражения
  // локальные переменные и т. д.
};
```

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается. Таким образом, такой метод создания позволяет инкапсулировать код, который создаёт отдельный объект, но без возможности его повторного использования.

Проверка на вызов в режиме конструктора: new.target

Продвинутая возможность

Данный метод используется очень редко. Вы можете пропустить эту секцию, если не хотите углубляться в детали языка.

Используя специальное свойство `new.target` внутри функции, мы можем проверить, вызвана ли функция при помощи оператора `new` или без него.

В случае, если функция вызвана при помощи `new`, то в `new.target` будет сама функция, в противном случае `undefined`.

```
function User() {
  alert(new.target);
}

// без "new":


// с "new":
User(); // undefined
new User(); // function User { ... }
```

Это можно использовать, чтобы отличить обычный вызов от вызова «в режиме конструктора». В частности, вот так можно сделать, чтобы функцию можно было вызывать как с, так и без `new`:

```
function User(name) {
  if (!new.target) { // в случае, если вы вызвали без оператора new
    return new User(name); // ...добавим оператор new за вас
  }

  this.name = name;
}

let vasya = User("Вася"); // переадресовывает вызовы на new User
alert(vasya.name); // Вася
```

Такой подход иногда используется в библиотеках для создания более гибкого синтаксиса, который позволяет разработчикам вызывать функции при помощи оператора `new` или без него.

Впрочем, это не очень хорошая практика, так как

отсутствие `new` может ввести разработчика в заблуждение. С оператором `new` мы точно знаем, что в итоге будет создан новый объект.

Возврат значения из конструктора return

Обычно конструкторы ничего не возвращают явно. Их задача – записать все необходимое в `this`, который в итоге станет результатом.

Но если `return` всё же есть, то применяется простое правило:

> При вызове `return` с объектом, будет возвращён объект, а не `this`.
>
> При вызове `return` с примитивным значением, примитивное значение будет отброшено.

Другими словами, `return` с объектом возвращает объект, в любом другом случае конструктор вернёт `this`.

В примере ниже `return` возвращает объект вместо `this`:

```
function BigUser() {

  this.name = "Вася";

  return { name: "Godzilla" };  // <-- возвращает этот объект
}

alert( new BigUser().name );  // Godzilla, получили этот
объект
```

А вот пример с пустым `return` (или мы могли бы поставить примитив после `return`, неважно)

```
function SmallUser() {
```

```
  this.name = "Вася";

  return; // <-- возвращает this
}

alert( new SmallUser().name );  // Вася
```

Обычно у конструкторов отсутствует `return`. В данном блоке мы упомянули особое поведение с возвращаемыми объектами, чтобы не оставлять пробелов в изучении языка.

Отсутствие скобок

Кстати, мы можем не ставить скобки после `new`, если вызов конструктора идёт без аргументов.

```
let user = new User; // <-- без скобок
// то же, что и
let user = new User();
```

Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

Создание методов в конструкторе

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.

В `this` мы можем добавлять не только свойства, но и методы.

Например, в примере ниже, `new User(name)` создаёт объект с данным именем `name` и методом `sayHi`:

```
function User(name) {
  this.name = name;

  this.sayHi = function() {
    alert( "Меня зовут: " + this.name );
  };
}



/*
vasya = {
   name: "Вася",
   sayHi: function() { ... }
}
*/
let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася
```

Для создания сложных объектов есть и более «продвинутый» синтаксис – классы, которые мы разберём позже.

Итого

> Функции-конструкторы или просто конструкторы являются обычными функциями, именовать которые следует с заглавной буквы.
>
> Конструкторы следует вызывать при помощи оператора `new`. Такой вызов создаёт пустой `this` в начале выполнения и возвращает заполненный в конце.

Мы можем использовать конструкторы для создания множества похожих объектов.

JavaScript предоставляет функции-конструкторы для множества встроенных объектов языка: например, `Date`, `Set` и других, которые нам ещё предстоит изучить.

Объекты, мы к ним ещё вернёмся!

В этой главе мы рассмотрели базовые принципы объектов и конструкторов. Данная информация необходима нам для дальнейшего изучения типов данных и функций. Как только мы с ними разберёмся, мы вернёмся к объектам для более детального изучения в главах Прототипы, наследование и Классы.

Задачи

Две функции - один объект

важность: 2

Возможно ли создать функции `A` и `B` в примере ниже, где объекты равны `new A()==new B()`?

```
function A() { ... }
function B() { ... }

let a = new A;
let b = new B;

alert( a == b ); // true
```

Если да – приведите пример вашего кода.

К решению

Создание калькулятора при помощи конструктора

важность: 5

Создайте функцию-конструктор `Calculator`, который создаёт объекты с тремя методами:

- `read()` запрашивает два значения при помощи `prompt` и сохраняет их значение в свойствах объекта.
- `sum()` возвращает сумму введённых свойств.
- `mul()` возвращает произведение введённых свойств.

Например:

```
let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

Запустить демо

Открыть песочницу с тестами для задачи.

К решению

Создаём Accumulator

важность: 5

Напишите функцию-конструктор `Accumulator(startingValue)`.

Объект, который она создаёт, должен уметь следующее:

- Хранить «текущее значение» в свойстве `value`. Начальное значение устанавливается в аргументе конструктора `startingValue`.
- Метод `read()` использует `prompt` для получения

числа и прибавляет его к свойству `value`.

Таким образом, свойство `value` является текущей суммой всего, что ввёл пользователь при вызовах метода `read()`, с учётом начального значения `startingValue`.

Ниже вы можете посмотреть работу кода:

```
let accumulator = new Accumulator(1); // начальное значение 1

accumulator.read(); // прибавит ввод prompt к текущему значению
accumulator.read(); // прибавит ввод prompt к текущему значению

alert(accumulator.value); // выведет сумму этих значений
```

Запустить демо

Открыть песочницу с тестами для задачи.

К решению

Опциональная цепочка '?.'

> **Новая возможность**
>
> Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Опциональная цепочка `?.` — это безопасный способ доступа к свойствам вложенных объектов, даже если какое-либо из промежуточных свойств не существует.

Проблема «несуществующего свойства»

Если вы только начали читать учебник и изучать JavaScript, то, возможно, эта проблема вам пока незнакома, но она достаточно распространена.

Например, рассмотрим объекты для пользователей `user`. У большинства пользователей есть адрес `user.address` с улицей `user.address.street`, но некоторые адрес не указали.

В этом случае при попытке получить свойство `user.address.street` будет ошибка:

```
let user = {}; // пользователь без свойства address

alert(user.address.street); // ошибка!
```

Это нормальный результат, так работает JavaScript, но во многих реальных ситуациях удобнее было бы получать не ошибку, а просто `undefined` («нет улицы»).

Или ещё пример. В веб-разработке нам бывает нужно получить данные об HTML-элементе, который иногда может отсутствовать на странице:

```
// Произойдёт ошибка, если querySelector(...) равен null.
let html = document.querySelector('.my-element').innerHTML;
```

До появления `?.` в языке для решения подобных проблем использовался оператор `&&`.

Например:

```
let user = {}; // пользователь без адреса

alert( user && user.address && user.address.street ); //
undefined (без ошибки)
```

Использование логического И со всей цепочкой свойств гарантирует, что все они существуют (а если нет – вычисление прекращается), но это довольно длинная и громоздкая конструкция.

Опциональная цепочка

Опциональная цепочка `?.` останавливает вычисление и возвращает `undefined`, если часть перед `?.` имеет значение `undefined` или `null`.

Для краткости в этой статье мы будем говорить о значении, что оно «существует», если оно отличается от `null` или `undefined`.

Вот безопасный способ обратиться к свойству `user.address.street`:

```
let user = {}; // пользователь без адреса

alert( user?.address?.street ); // undefined (без ошибки)
```

Чтение адреса с помощью конструкции `user?.address` выполняется без ошибок, даже если объекта `user` не существует:

```
let user = null;

alert( user?.address ); // undefined
alert( user?.address.street ); // undefined
```

Обратите внимание, что синтаксис `?.` делает необязательным только свойство перед ним, а не какое-

либо последующее.

В приведённом выше примере конструкция `user?.` допускает, что переменная `user` может содержать `null/undefined`.

С другой стороны, если объект `user` существует, то в нём должно быть свойство `user.address`, иначе выполнение `user?.address.street` вызовет ошибку из-за второй точки.

Не злоупотребляйте опциональной цепочкой

Используйте `?.` только тогда, когда допускаете ситуацию, что значение перед ним не существует.

Например, если по нашей логике объект `user` точно существует, но его свойство `address` является необязательным, то предпочтительнее использовать следующую конструкцию: `user.address?.street`.

Тогда если переменная `user` по ошибке окажется пустой, мы увидим программную ошибку и исправим это.

Переменная перед `?.` должна быть объявлена

Если переменной `user` вообще не существует, то выражение `user?.anything` выдаст ошибку:

```
// ReferenceError: user is not defined
```

```
user?.address;
```

Объявление переменной (например `let/const/var user`) обязательно должно быть. Опциональная цепочка работает только с существующими переменными.

Сокращённое вычисление

Как уже говорилось, `?.` немедленно останавливает вычисление, если левой части не существует.

Таким образом, последующие вызовы функций или операции не будут выполнены.

Например:

```
let user = null;
let x = 0;

user?.sayHi(x++); // нет user, поэтому до x++ вычисление не дойдет

alert(x); // 0, значение не было увеличено на единицу
```

Другие варианты применения: ?.(), ?.[]

Опциональная цепочка `?.` — это не оператор, а специальная синтаксическая конструкция, которая также работает с функциями и квадратными скобками.

Например, `?.()` используется для вызова потенциально несуществующей функции.

В следующем примере не у всех пользователей есть метод `admin`:

```
let user1 = {
  admin() {
    alert("Я администратор");
  }
}

let user2 = {};

user1.admin?.(); // Я администратор
user2.admin?.();
```

В обоих вызовах сначала используем точку (`user1.admin`), чтобы получить свойство `admin`, потому что объект пользователя точно существует, к нему можно обратиться без какой-либо ошибки.

Затем уже `?.()` проверяет левую часть: если функция `admin` существует, то она выполнится (это так для `user1`). Иначе (для `user2`) вычисление остановится без ошибок.

Также существует синтаксис `?.[]`, если значение свойства требуется получить с помощью квадратных скобок `[]`, а не через точку `.`. Как и в остальных случаях, такой способ позволяет защититься от ошибок при доступе к свойству объекта, которого может не быть.

```
let user1 = {
  firstName: "Иван"
};

let user2 = null; // Представим, что пользователь не
авторизован

let key = "firstName";

alert( user1?.[key] ); // Иван
alert( user2?.[key] ); // undefined

alert( user1?.[key]?.something?.not?.existing); // undefined
```

Кроме этого, `?.` можно совместно использовать с `delete`:

```
delete user?.name; // Удалить user.name, если пользователь существует
```

Можно использовать `?.` для безопасного чтения и удаления, но не для записи

Опциональная цепочка `?.` не имеет смысла в левой части присваивания.

Например:

```
let user;

user?.name = "John"; // Ошибка, это не сработает
// это по сути то же самое что undefined = "John"
```

Она недостаточно «умна» для этого.

Итого

Синтаксис опциональной цепочки `?.` имеет три формы:

1. `obj?.prop` – возвращает `obj.prop`, если существует `obj`, и `undefined` в противном случае.
2. `obj?.[prop]` – возвращает `obj[prop]`, если существует `obj`, и `undefined` в противном случае.
3. `obj.method?.()` – вызывает `obj.method()`, если существует `obj.method`, в противном случае возвращает `undefined`.

Как мы видим, все они просты и понятны в использовании. `?.` проверяет левую часть выражения на равенство `null/undefined`, и продолжает дальнейшее вычисление, только если это не так.

Цепочка `?.` позволяет без возникновения ошибок обратиться к вложенным свойствам.

Тем не менее, нужно разумно использовать `?.` — только там, где это уместно, если допустимо, что левая часть не существует. Чтобы таким образом не скрывать возможные ошибки программирования.

Тип данных Symbol

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки или символы. Ни числа, ни логические значения не подходят, разрешены только эти два типа данных.

До сих пор мы видели только строки. Теперь давайте разберём символы, увидим, что хорошего они нам дают.

Символы

«Символ» представляет собой уникальный идентификатор.

Создаются новые символы с помощью функции `Symbol()`:

```
// Создаём новый символ – id
let id = Symbol();
```

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода:

```
// Создаём символ id с описанием (именем) "id"
let id = Symbol("id");
```

Символы гарантированно уникальны. Даже если мы создадим множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет.

Например, вот два символа с одинаковым описанием – но они не равны:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

Если вы знаете Ruby или какой-то другой язык программирования, в котором есть своего рода «символы» – пожалуйста, будьте внимательны. Символы в JavaScript имеют свои особенности, и не стоит думать о них, как о символах в Ruby или в других языках.

Символы не преобразуются автоматически в строки

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция `alert` принимает практически любое значение, автоматически преобразовывает его в строку, а затем выводит это значение, не сообщая об ошибке. Символы же особенные и не преобразуются автоматически.

К примеру, `alert` ниже выдаст ошибку:

```
let id = Symbol("id");
alert(id); // TypeError: Cannot convert
a Symbol value to a string
```

Это – языковая «защита» от путаницы, ведь

строки и символы – принципиально разные типы данных и не должны неконтролируемо преобразовываться друг в друга.

Если же мы действительно хотим вывести символ с помощью `alert`, то необходимо явно преобразовать его с помощью метода `.toString()`, вот так:

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), теперь работает
```

Или мы можем обратиться к свойству `symbol.description`, чтобы вывести только описание:

```
let id = Symbol("id");
alert(id.description); // id
```

«Скрытые» свойства

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы.

Например, мы работаем с объектами `user`, которые принадлежат стороннему коду. Мы хотим добавить к ним идентификаторы.

Используем для этого символьный ключ:

```
let user = {
  name: "Вася"
};

let id = Symbol("id");
```

```
user[id] = 1;

alert( user[id] ); // мы можем получить доступ к данным по ключу-символу
```

Почему же лучше использовать `Symbol("id")`, а не строку `"id"`?

Так как объект `user` принадлежит стороннему коду, и этот код также работает с ним, то нам не следует добавлять к нему какие-либо поля. Это небезопасно. Но к символу сложно нечаянно обратиться, сторонний код вряд ли его вообще увидит, и, скорее всего, добавление поля к объекту не вызовет никаких проблем.

Кроме того, предположим, что другой скрипт для каких-то своих целей хочет записать собственный идентификатор в объект `user`. Этот скрипт может быть какой-то JavaScript-библиотекой, абсолютно не связанной с нашим скриптом.

Сторонний код может создать для этого свой символ `Symbol("id")`:

```
// ...
let id = Symbol("id");

user[id] = "Их идентификатор";
```

Конфликта между их и нашим идентификатором не будет, так как символы всегда уникальны, даже если их имена совпадают.

А вот если бы мы использовали строку `"id"` вместо символа, то тогда *был бы* конфликт:

```
let user = { name: "Вася" };

// Объявляем в нашем скрипте свойство "id"
user.id = "Наш идентификатор";

// ...другой скрипт тоже хочет свой идентификатор...
```

```
user.id = "Их идентификатор"
// Ой! Свойство перезаписано сторонней библиотекой!
```

Символы в литеральном объекте

Если мы хотим использовать символ при литеральном объявлении объекта `{...}`, его необходимо заключить в квадратные скобки.

Вот так:

```
let id = Symbol("id");

let user = {
  name: "Вася",

};
   [id]: 123 // просто "id: 123" не сработает
```

Это вызвано тем, что нам нужно использовать значение переменной `id` в качестве ключа, а не строку «id».

Символы игнорируются циклом for…in

Свойства, чьи ключи – символы, не перебираются циклом `for..in`.

Например:

```
let id = Symbol("id");
let user = {
  name: "Вася",
  age: 30,
  [id]: 123
};



// хотя прямой доступ по символу работает
alert( "Напрямую: " + user[id] );
for (let key in user) alert(key); // name, age
(свойства с ключом-символом нет среди
перечисленных)
```

Это – часть общего принципа «сокрытия символьных свойств». Если другая библиотека или скрипт будут работать с нашим объектом, то при переборе они не получат ненароком наше символьное свойство. `Object.keys(user)` также игнорирует символы.

А вот Object.assign, в отличие от цикла `for..in`, копирует и строковые, и символьные свойства:

```
let id = Symbol("id");
let user = {
  [id]: 123
};

let clone = Object.assign({}, user);

alert( clone[id] ); // 123
```

Здесь нет никакого парадокса или противоречия. Так и задумано. Идея заключается в том, что, когда мы клонируем или объединяем объекты, мы обычно хотим скопировать *все* свойства (включая такие свойства с ключами-символами, как, например, `id` в примере выше).

Глобальные символы

Итак, как мы видели, обычно все символы уникальны, даже если их имена совпадают. Но иногда мы наоборот хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к символу `"id"`, подразумевая именно одно и то же свойство.

Для этого существует *глобальный реестр символов*. Мы можем создавать в нём символы и обращаться к ним позже, и при каждом обращении нам гарантированно будет возвращаться один и тот же символ.

Для чтения (или, при отсутствии, создания) символа из

реестра используется вызов `Symbol.for(key)`.

Он проверяет глобальный реестр и, при наличии в нём символа с именем `key`, возвращает его, иначе же создаётся новый символ `Symbol(key)` и записывается в реестр под ключом `key`.

Например:

```
// читаем символ из глобального реестра и записываем его в переменную
let id = Symbol.for("id"); // если символа не существует, он будет создан

// читаем его снова в другую переменную (возможно, из другого места кода)
let idAgain = Symbol.for("id");

// проверяем -- это один и тот же символ
alert( id === idAgain ); // true
```

Символы, содержащиеся в реестре, называются *глобальными символами*. Если вам нужен символ, доступный везде в коде – используйте глобальные символы.

Похоже на Ruby

В некоторых языках программирования, например, Ruby, на одно имя (описание) приходится один символ, и не могут существовать разные символы с одинаковым именем.

В JavaScript, как мы видим, это утверждение верно только для глобальных символов.

Symbol.keyFor

Для глобальных символов, кроме `Symbol.for(key)`,

который ищет символ по имени, существует обратный метод: `Symbol.keyFor(sym)`, который, наоборот, принимает глобальный символ и возвращает его имя.

К примеру:

```
// получаем символ по имени
let sym = Symbol.for("name");
let sym2 = Symbol.for("id");

// получаем имя по символу
alert( Symbol.keyFor(sym) ); // name
alert( Symbol.keyFor(sym2) ); // id
```

Внутри метода `Symbol.keyFor` используется глобальный реестр символов для нахождения имени символа. Так что этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт `undefined`.

Впрочем, для любых символов доступно свойство `description`.

Например:

```
let globalSymbol = Symbol.for("name");
let localSymbol = Symbol("name");

alert( Symbol.keyFor(globalSymbol) ); // name, глобальный
символ
alert( Symbol.keyFor(localSymbol) ); // undefined для
неглобального символа

alert( localSymbol.description ); // name
```

Системные символы

Существует множество «системных» символов, использующихся внутри самого JavaScript, и мы можем использовать их, чтобы настраивать различные аспекты поведения объектов.

Эти символы перечислены в спецификации в таблице Well-known symbols:

- `Symbol.hasInstance`
- `Symbol.isConcatSpreadable`
- `Symbol.iterator`
- `Symbol.toPrimitive`
- …и так далее.

В частности, `Symbol.toPrimitive` позволяет описать правила для объекта, согласно которым он будет преобразовываться к примитиву. Мы скоро увидим его применение.

С другими системными символами мы тоже скоро познакомимся, когда будем изучать соответствующие возможности языка.

Итого

Символ (symbol) – примитивный тип данных, использующийся для создания уникальных идентификаторов.

Символы создаются вызовом функции `Symbol()`, в которую можно передать описание (имя) символа.

Даже если символы имеют одно и то же имя, это – разные символы. Если мы хотим, чтобы одноимённые символы были равны, то следует использовать глобальный реестр: вызов `Symbol.for(key)` возвращает (или создаёт) глобальный символ с `key` в качестве имени. Многократные вызовы команды `Symbol.for` с одним и тем же аргументом возвращают один и тот же символ.

Символы имеют два основных варианта использования:

1. «Скрытые» свойства объектов. Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в `for..in`, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования.

   Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, но что другие видеть не должны.

2. Существует множество системных символов, используемых внутри JavaScript, доступных как `Symbol.*`. Мы можем использовать их, чтобы изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать `Symbol.iterator` для итераторов, `Symbol.toPrimitive` для настройки преобразования объектов в примитивы и так далее.

Технически символы скрыты не на 100%. Существует встроенный метод Object.getOwnPropertySymbols(obj) – с его помощью можно получить все свойства объекта с ключами-символами. Также существует метод Reflect.ownKeys(obj), который возвращает *все* ключи объекта, включая символьные. Так что они не совсем спрятаны. Но большинство библиотек, встроенных методов и синтаксических конструкций не используют эти методы.

Преобразование объектов в примитивы

Что произойдёт, если сложить два объекта `obj1 + obj2`, вычесть один из другого `obj1 - obj2` или вывести их на

экран, воспользовавшись `alert(obj)`?

В этом случае объекты сначала автоматически преобразуются в примитивы, а затем выполняется операция.

В главе Преобразование типов мы видели правила для численных, строковых и логических преобразований. Но обделили вниманием объекты. Теперь, поскольку мы уже знаем о методах объектов и символах, можно исправить это упущение.

1. Все объекты в логическом контексте являются `true`. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда мы вычитаем объекты или выполняем математические операции. Например, объекты `Date` (мы рассмотрим их в статье Дата и время) могут вычитаться, и результатом `date1 - date2` будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, когда мы выводим объект `alert(obj)`, а также в других случаях, когда объект используется как строка.

Преобразование к примитивам

Мы можем тонко настраивать строковые и численные преобразования, используя специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в спецификации:

`"string"`

Для преобразования объекта к строке, когда операция ожидает получить строку, например `alert`:

```
// вывод
alert(obj);

// используем объект в качестве имени свойства
anotherObj[obj] = 123;
```

"number"

Для преобразования объекта к числу, в случае математических операций:

```
// явное преобразование
let num = Number(obj);

// математическое (исключая бинарный оператор "+")
let n = +obj; // унарный плюс
let delta = date1 - date2;

// сравнения больше/меньше
let greater = user1 > user2;
```

"default"

Происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, бинарный плюс + может работать с обоими типами: строками (объединять их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или когда происходит сравнение объектов с помощью нестрогого равенства == со строкой, числом или символом, и неясно, какое преобразование должно быть выполнено.

```
// бинарный плюс
let total = car1 + car2;

// obj == string/number/symbol
if (user == 1) { ... };
```

Оператор больше/меньше <> также может работать как со строками, так и с числами. Однако, по историческим причинам он использует хинт «number», а не «default».

На практике все встроенные объекты, исключая `Date` (мы познакомимся с ним чуть позже), реализуют `"default"` преобразования тем же способом, что и `"number"`. И нам следует поступать так же.

Обратите внимание, что существуют лишь три варианта хинтов. Всё настолько просто. Не существует хинта со значением «boolean» (все объекты являются `true` в логическом контексте) или каких-либо ещё. И если мы считаем `"default"` и `"number"` одинаковыми, как большинство встроенных объектов, то остаются всего два варианта преобразований.

В процессе преобразования движок JavaScript пытается найти и вызвать три следующих метода объекта:

1. Вызывает `obj[Symbol.toPrimitive](hint)` – метод с символьным ключом `Symbol.toPrimitive` (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен `"string"`

   пытается вызвать `obj.toString()`, а если его нет, то `obj.valueOf()`, если он существует.
4. В случае, если хинт равен `"number"` или `"default"`

   пытается вызвать `obj.valueOf()`, а если его нет, то `obj.toString()`, если он существует.

Symbol.toPrimitive

Начнём с универсального подхода – символа `Symbol.toPrimitive`: метод с таким названием (если есть) используется для всех преобразований:

```
obj[Symbol.toPrimitive] = function(hint) {
  // должен вернуть примитивное значение
  // hint равно чему-то одному из: "string", "number" или
```

```
"default"
};
```

Для примера используем его в реализации объекта `user`:

```
let user = {
  name: "John",
  money: 1000,

  [Symbol.toPrimitive](hint) {
    alert(`hint: ${hint}`);
    return hint == "string" ? `{name: "${this.name}"}` :
this.money;
  }
};

// демонстрация результатов преобразований:
alert(user); // hint: string -> {name: "John"}
alert(+user); // hint: number -> 1000
alert(user + 500); // hint: default -> 1500
```

Как мы видим из кода, `user` преобразовывается либо в информативную читаемую строку, либо в денежный счёт в зависимости от значения хинта. Единственный метод `user[Symbol.toPrimitive]` смог обработать все случаи преобразований.

Методы toString/valueOf

Методы `toString` и `valueOf` берут своё начало с древних времён. Они не символы, так как в то время символов ещё не существовало, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода `Symbol.toPrimitive`, движок JavaScript пытается найти эти методы и вызвать их следующим образом:

- `toString -> valueOf` для хинта со значением

«string».

`valueOf -> toString` – в ином случае.

Для примера, используем их в реализации всё того же объекта `user`. Воспроизведём его поведение комбинацией методов `toString` и `valueOf`:

```
let user = {
  name: "John",
  money: 1000,

  // для хинта равного "string"
  toString() {
    return `{name: "${this.name}"}`;
  },

  // для хинта равного "number" или "default"
  valueOf() {
    return this.money;
  }

};

alert(user); // toString -> {name: "John"}
alert(+user); // valueOf -> 1000
alert(user + 500); // valueOf -> 1500
```

Как видим, получилось то же поведение, что и в предыдущем примере с `Symbol.toPrimitive`.

Довольно часто мы хотим описать одно «универсальное» преобразование объекта к примитиву для всех ситуаций. Для этого достаточно создать один `toString`:

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user); // toString -> John
alert(user + 500); // toString -> John500
```

В отсутствие `Symbol.toPrimitive` и `valueOf`, `toString` обработает все случаи преобразований к примитивам.

Возвращаемые типы

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно требуемый «хинтом» тип примитива.

Нет обязательного требования, чтобы `toString()` возвращал именно строку, или чтобы метод `Symbol.toPrimitive` возвращал именно число для хинта «number».

Единственное обязательное требование: методы должны возвращать примитив, а не объект.

Историческая справка

По историческим причинам, если `toString` или `valueOf` вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало).

Метод `Symbol.toPrimitive`, напротив, *обязан* возвращать примитив, иначе будет ошибка.

Последующие операции

Операция, инициировавшая преобразование, получает примитив и затем продолжает работу с ним, производя дальнейшие преобразования, если это необходимо.

Например:

Математические операции, исключая бинарный плюс, преобразуют примитив к числу:

```
let obj = {
  // toString обрабатывает все преобразования в случае
отсутствия других методов
  toString() {
    return "2";
  }
};

alert(obj * 2); // 4, объект был преобразован к примитиву
"2", затем умножение сделало его числом
```

Бинарный плюс + в аналогичном случае сложит строки:

```
let obj = {
  toString() {
    return "2";
  }
};

alert(obj + 2); // 22 (преобразование к примитиву вернуло
строку => конкатенация)
```

Итого

Преобразование объектов в примитивы вызывается автоматически многими встроенными функциями и операторами, которые ожидают примитив в качестве аргумента.

Существует всего 3 типа преобразований (хинтов):

- `"string"` (для `alert` и других операций, которым нужна строка)
- `"number"` (для математических операций)
- `"default"` (для некоторых операций)

В спецификации явно указано, какой хинт должен использовать каждый оператор. И существует совсем немного операторов, которые не знают, что ожидать, и используют хинт со значением `"default"`. Обычно для встроенных объектов хинт `"default"` обрабатывается так же, как `"number"`. Таким образом, последние два очень часто объединяют вместе.

Алгоритм преобразований к примитивам следующий:

1. Сначала вызывается метод `obj[Symbol.toPrimitive](hint)`, если он существует.
2. Иначе, если хинт равен `"string"`

   происходит попытка вызвать `obj.toString()`, затем `obj.valueOf()`, смотря что есть.
4. Иначе, если хинт равен `"number"` или `"default"`

   происходит попытка вызвать `obj.valueOf()`, затем `obj.toString()`, смотря что есть.

На практике довольно часто достаточно реализовать только `obj.toString()` как «универсальный» метод для всех типов преобразований, возвращающий «читаемое» представление объекта, достаточное для логирования или отладки.

Типы данных

Больше структур данных и более глубокое изучение типов.

Методы у примитивов

JavaScript позволяет нам работать с примитивными типами данных – строками, числами и т.д., как будто они являются объектами. У них есть и методы. Мы изучим их позже, а сначала разберём, как это всё работает, потому что, конечно, примитивы – не объекты.

Давайте взглянем на ключевые различия между примитивами и объектами.

Примитив

> Это – значение «примитивного» типа.
>
> Есть 7 примитивных типов: `string`, `number`, `boolean`, `symbol`, `null`, `undefined` и `bigint`.

Объект

> Может хранить множество значений как свойства.
>
> Объявляется при помощи фигурных скобок `{}`, например: `{name: "Рома", age: 30}`. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что мы можем хранить функцию как одно из свойств объекта.

```
let roma = {
  name: "Рома",
  sayHi: function() {
```

```
    alert("Привет, дружище!");
  }
};

roma.sayHi(); // Привет, дружище!
```

Здесь мы создали объект `roma` с методом `sayHi`.

Существует множество встроенных объектов. Например, те, которые работают с датами, ошибками, HTML-элементами и т.д. Они имеют различные свойства и методы.

Однако у этих возможностей есть обратная сторона!

Объекты «тяжелее» примитивов. Они нуждаются в дополнительных ресурсах для поддержания внутренней структуры.

Примитив как объект

Вот парадокс, с которым столкнулся создатель JavaScript:

> Есть много всего, что хотелось бы сделать с примитивами, такими как строка или число. Было бы замечательно, если бы мы могли работать с ними через вызовы методов.
>
> Примитивы должны быть лёгкими и быстрыми.

Выбранное решение, хотя выглядит оно немного неуклюже:

1. Примитивы остаются примитивами. Одно значение, как и хотелось.
2. Язык позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов.
3. Чтобы это работало, при таком доступе создаётся специальный «объект-обёртка», который

предоставляет нужную функциональность, а после удаляется.

Каждый примитив имеет свой собственный «объект-обёртку», которые называются: `String`, `Number`, `Boolean` и `Symbol`. Таким образом, они имеют разный набор методов.

К примеру, существует метод str.toUpperCase(), который возвращает строку в верхнем регистре.

Вот, как он работает:

```
let str = "Привет";

alert( str.toUpperCase() ); // ПРИВЕТ
```

Очень просто, не правда ли? Вот, что на самом деле происходит в `str.toUpperCase()`:

1. Строка `str` – примитив. В момент обращения к его свойству, создаётся специальный объект, который знает значение строки и имеет такие полезные методы, как `toUpperCase()`.
2. Этот метод запускается и возвращает новую строку (показывается в `alert`).
3. Специальный объект удаляется, оставляя только примитив `str`.

Получается, что примитивы могут предоставлять методы, и в то же время оставаться «лёгкими».

Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта. Однако, он всё же должен придерживаться спецификаций и работать так, как будто он его создаёт.

Число имеет собственный набор методов. Например, toFixed(n) округляет число до n знаков после запятой.

```
let n = 1.23456;

alert( n.toFixed(2) ); // 1.23
```

Более подробно с различными свойствами и методами мы познакомимся в главах Числа и Строки.

Конструкторы `String/Number/Boolean` предназначены только для внутреннего пользования

Некоторые языки, такие как Java, позволяют явное создание «объектов-обёрток» для примитивов при помощи такого синтаксиса как `new Number(1)` или `new Boolean(false)`.

В JavaScript, это тоже возможно по историческим причинам, но очень **не рекомендуется**. В некоторых местах последствия могут быть катастрофическими.

Например:

```
alert( typeof 0 ); // "число"

alert( typeof new Number(0) ); // "object"!
```

Объекты в `if` всегда дают `true`, так что в нижеприведённом примере будет показан `alert`:

```
let zero = new Number(0);

if (zero) {
  // zero возвращает "true", так как является
объектом
  alert( "zero имеет «истинное» значение?!?" );
}
```

С другой стороны, использование функций `String/Number/Boolean` без оператора `new` – вполне разумно и полезно. Они превращают значение в соответствующий примитивный тип: в строку, в число, в булевый тип.

К примеру, следующее вполне допустимо:

```
let num = Number("123"); // превращает строку в
число
```

null/undefined не имеют методов

Особенные примитивы `null` и `undefined` являются исключениями. У них нет соответствующих «объектов-обёрток», и они не имеют никаких методов. В некотором смысле, они «самые примитивные».

Попытка доступа к свойствам такого значения возвратит ошибку:

```
alert(null.test); // ошибка
```

Итого

Все примитивы, кроме `null` и `undefined`, предоставляют множество полезных методов. Мы познакомимся с ними поближе в следующих главах.

Формально эти методы работают с помощью временных объектов, но движки JavaScript внутренне очень хорошо оптимизируют этот процесс, так что их вызов не требует много ресурсов.

Задачи

Можно ли добавить свойство строке?

важность: 5

Взгляните на следующий код:

```
let str = "Привет";

str.test = 5;

alert(str.test);
```

Как вы думаете, это сработает? Что выведется на экран?

К решению

Числа

В современном JavaScript существует два типа чисел:

1. Обычные числа в JavaScript хранятся в 64-битном формате IEEE-754, который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers). Это числа, которые мы будем использовать чаще всего. Мы поговорим о них в этой главе.
2. `BigInt` числа дают возможность работать с целыми числами произвольной длины. Они нужны достаточно редко и используются в случаях, когда необходимо работать со значениями более чем 2^{53} или менее чем

-2^{53}. Так как `BigInt` числа нужны достаточно редко, мы рассмотрим их в отдельной главе BigInt.

В данной главе мы рассмотрим только первый тип чисел: числа типа `number`. Давайте глубже изучим, как с ними работать в JavaScript.

Способы записи числа

Представьте, что нам надо записать число 1 миллиард. Самый очевидный путь:

```
let billion = 1000000000;
```

Но в реальной жизни мы обычно опускаем запись множества нулей, так как можно легко ошибиться. Укороченная запись может выглядеть как `"1млрд"` или `"7.3млрд"` для 7 миллиардов 300 миллионов. Такой принцип работает для всех больших чисел.

В JavaScript можно использовать букву `"e"`, чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

```
let billion = 1e9;  // 1 миллиард, буквально: 1 и 9 нулей

alert( 7.3e9 );  // 7.3 миллиардов (7,300,000,000)
```

Другими словами, `"e"` производит операцию умножения числа на 1 с указанным количеством нулей.

```
1e3 = 1 * 1000
1.23e6 = 1.23 * 1000000
```

Сейчас давайте запишем что-нибудь очень маленькое. К примеру, 1 микросекунду (одна миллионная секунды):

```
let ms = 0.000001;
```

Записать микросекунду в укороченном виде нам поможет `"e"`.

```
let ms = 1e-6; // шесть нулей, слева от 1
```

Если мы подсчитаем количество нулей `0.000001`, их будет 6. Естественно, верная запись `1e-6`.

Другими словами, отрицательное число после `"e"` подразумевает деление на 1 с указанным количеством нулей:

```
// 1 делится на 1 с 3 нулями
1e-3 = 1 / 1000 (=0.001)

// 1.23 делится на 1 с 6 нулями
1.23e-6 = 1.23 / 1000000 (=0.00000123)
```

Шестнадцатеричные, двоичные и восьмеричные числа

Шестнадцатеричные числа широко используются в JavaScript для представления цветов, кодировки символов и многого другого. Естественно, есть короткий стиль записи: `0x`, после которого указывается число.

Например:

```
alert( 0xff ); // 255
alert( 0xFF ); // 255 (то же самое, регистр не имеет значения)
```

Не так часто используются двоичные и восьмеричные числа, но они также поддерживаются `0b` для двоичных и `0o` для восьмеричных:

```
let a = 0b11111111; // бинарная форма записи числа 255
let b = 0o377; // восьмеричная форма записи числа 255
```

```
alert( a == b ); // true, с двух сторон число 255
```

Есть только 3 системы счисления с такой поддержкой. Для других систем счисления мы рекомендуем использовать функцию `parseInt` (рассмотрим позже в этой главе).

toString(base)

Метод `num.toString(base)` возвращает строковое представление числа `num` в системе счисления `base`.

Например:

```
let num = 255;

alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

`base` может варьироваться от `2` до `36` (по умолчанию `10`).

Часто используемые:

base=16 — для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть `0..9` или `A..F`.

base=2 — обычно используется для отладки побитовых операций, цифры `0` или `1`.

base=36 — максимальное основание, цифры могут быть `0..9` или `A..Z`. То есть, используется весь латинский алфавит для представления числа. Забавно, но можно использовать `36`-разрядную систему счисления для получения короткого представления большого числового идентификатора. К примеру, для создания короткой ссылки. Для этого просто преобразуем его в `36`-разрядную систему счисления:

```
alert( 123456..toString(36) ); // 2n9c
```

Две точки для вызова метода

Внимание! Две точки в `123456..toString(36)` это не опечатка. Если нам надо вызвать метод непосредственно на числе, как `toString` в примере выше, то нам надо поставить две точки `..` после числа.

Если мы поставим одну точку: `123456.toString(36)`, тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод.

Также можно записать как `(123456).toString(36)`.

Округление

Одна из часто используемых операций при работе с числами – это округление.

В JavaScript есть несколько встроенных функций для работы с округлением:

`Math.floor`

Округление в меньшую сторону: `3.1` становится `3`, а `-1.1` — `-2`.

`Math.ceil`

Округление в большую сторону: `3.1` становится `4`, а

`-1.1` — `-1`.

`Math.round`

Округление до ближайшего целого: `3.1` становится `3`, `3.6` — `4`, а `-1.1` — `-1`.

`Math.trunc` (не поддерживается в Internet Explorer)

Производит удаление дробной части без округления: `3.1` становится `3`, а `-1.1` — `-1`.

Ниже представлена таблица с различиями между функциями округления:

	`Math.floo`	`Math.cei`	`Math.roun`	`Math.trunc`
`3.1`	`3`	`4`	`3`	`3`
`3.6`	`3`	`4`	`4`	`3`
`-1.1`	`-2`	`-1`	`-1`	`-1`
`-1.6`	`-2`	`-1`	`-2`	`-1`

Эти функции охватывают все возможные способы обработки десятичной части. Что если нам надо округлить число до `n-ого` количества цифр в дробной части?

Например, у нас есть `1.2345` и мы хотим округлить число до 2-х знаков после запятой, оставить только `1.23`.

Есть два пути решения:

1. Умножить и разделить.

   Например, чтобы округлить число до второго знака после запятой, мы можем умножить число на `100`, вызвать функцию округления и разделить обратно.

```
let num = 1.23456;
```

```
alert( Math.floor(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

2. Метод toFixed(n) округляет число до `n` знаков после запятой и возвращает строковое представление результата.

```
let num = 12.34;
alert( num.toFixed(1) ); // "12.3"
```

Округляет значение до ближайшего числа, как в большую, так и в меньшую сторону, аналогично методу `Math.round`:

```
let num = 12.36;
alert( num.toFixed(1) ); // "12.4"
```

Обратите внимание, что результатом `toFixed` является строка. Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки:

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", добавлены нули, чтобы получить 5 знаков после запятой
```

Мы можем преобразовать полученное значение в число, используя унарный оператор `+` или `Number()`, пример с унарным оператором: `+num.toFixed(5)`.

Неточные вычисления

Внутри JavaScript число представлено в виде 64-битного формата IEEE-754. Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранение

знака.

Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

```
alert( 1e500 ); // Infinity
```

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности.

Посмотрите на это (неверное!) сравнение:

```
alert( 0.1 + 0.2 == 0.3 ); //
false
```

Да-да, сумма `0.1` и `0.2` не равна `0.3`.

Странно! Что тогда, если не `0.3`?

```
alert( 0.1 + 0.2 ); // 0.30000000000000004
```

Ой! Здесь гораздо больше последствий, чем просто некорректное сравнение. Представьте, вы делаете интернет-магазин и посетители формируют заказ из 2-х позиций за `$0.10` и `$0.20`. Итоговый заказ будет `$0.30000000000000004`. Это будет сюрпризом для всех.

Но почему это происходит?

Число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как `0.1`, `0.2`, которые выглядят довольно просто в десятичной системе счисления, на самом деле являются бесконечной дробью в двоичной форме.

Другими словами, что такое `0.1`? Это единица делённая на десять — `1/10`, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с

одной третьей: `1/3`, которая становится бесконечной дробью `0.33333(3)`.

Деление на `10` гарантированно хорошо работает в десятичной системе, но деление на `3` – нет. По той же причине и в двоичной системе счисления, деление на `2` обязательно сработает, а `1/10` становится бесконечной дробью.

В JavaScript нет возможности для хранения точных значений 0.1 или 0.2, используя двоичную систему, точно также, как нет возможности хранить одну третью в десятичной системе счисления.

Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют нам увидеть эту «крошечную потерю точности», но она существует.

Пример:

```
alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

И когда мы суммируем 2 числа, их «неточности» тоже суммируются.

Вот почему `0.1 + 0.2` – это не совсем `0.3`.

Не только в JavaScript

Справедливости ради заметим, что ошибка в точности вычислений для чисел с плавающей точкой сохраняется в любом другом языке, где используется формат IEEE 754, включая PHP, Java, C, Perl, Ruby.

Можно ли обойти проблему? Конечно, наиболее надёжный

способ — это округлить результат используя метод toFixed(n):

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // 0.30
```

Помните, что метод `toFixed` всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. Также это удобно для форматирования цен в интернет-магазине `$0.30`. В других случаях можно использовать унарный оператор `+`, чтобы преобразовать строку в число:

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

Также можно временно умножить число на 100 (или на большее), чтобы привести его к целому, выполнить математические действия, а после разделить обратно. Суммируя целые числа, мы уменьшаем погрешность, но она все равно появляется при финальном делении:

```
alert( (0.1 * 10 + 0.2 * 10) / 10 ); // 0.3
alert( (0.28 * 100 + 0.14 * 100) / 100); // 0.4200000000000001
```

Таким образом, метод умножения/деления уменьшает погрешность, но полностью её не решает.

Иногда можно попробовать полностью отказаться от дробей. Например, если мы в нашем интернет-магазине начнём использовать центы вместо долларов. Но что будет, если мы применим скидку 30%? На практике у нас не получится полностью избавиться от дроби. Просто используйте округление, чтобы отрезать «хвосты», когда надо.

Забавный пример

Попробуйте выполнить его:

```
// Привет! Я - число, растущее само по себе!
alert( 9999999999999999 ); // покажет
10000000000000000
```

Причина та же – потеря точности. Из 64 бит, отведённых на число, сами цифры числа занимают до 52 бит, остальные 11 бит хранят позицию десятичной точки и один бит – знак. Так что если 52 бит не хватает на цифры, то при записи пропадут младшие разряды.

Интерпретатор не выдаст ошибку, но в результате получится «не совсем то число», что мы и видим в примере выше. Как говорится: «как смог, так записал».

Два нуля

Другим забавным следствием внутреннего представления чисел является наличие двух нулей: `0` и `-0`.

Все потому, что знак представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая нуль.

В большинстве случаев это поведение незаметно, так как операторы в JavaScript воспринимают их одинаковыми.

Проверка: isFinite и isNaN

Помните эти специальные числовые значения?

`Infinity` (и `-Infinity`) — особенное численное значение, которое ведёт себя в точности как математическая бесконечность ∞.

`NaN` представляет ошибку.

Эти числовые значения принадлежат типу `number`, но они не являются «обычными» числами, поэтому есть функции для их проверки:

`isNaN(value)` преобразует значение в число и проверяет является ли оно `NaN`:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

Нужна ли нам эта функция? Разве не можем ли мы просто сравнить `=== NaN`? К сожалению, нет. Значение `NaN` уникально тем, что оно не является равным ни чему другому, даже самому себе:

```
alert( NaN === NaN ); // false
```

`isFinite(value)` преобразует аргумент в число и возвращает `true`, если оно является обычным числом, т.е. не `NaN/Infinity/-Infinity`:

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, потому что специальное значение: NaN
alert( isFinite(Infinity) ); // false, потому что специальное значение: Infinity
```

Иногда `isFinite` используется для проверки, содержится ли в строке число:

```
let num = +prompt("Enter a number", '');
```

```
// вернёт true всегда, кроме ситуаций, когда аргумент -
Infinity/-Infinity или не число
alert( isFinite(num) );
```

Помните, что пустая строка интерпретируется как `0` во всех числовых функциях, включая`isFinite`.

Сравнение `Object.is`

Существует специальный метод Object.is, который сравнивает значения примерно как `===`, но более надёжен в двух особых ситуациях:

1. Работает с `NaN`: `Object.is(NaN, NaN) === true`, здесь он хорош.
2. Значения `0` и `-0` разные: `Object.is(0, -0) === false`, это редко используется, но технически эти значения разные.

Во всех других случаях `Object.is(a, b)` идентичен `a === b`.

Этот способ сравнения часто используется в спецификации JavaScript. Когда внутреннему алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует `Object.is` (Определение SameValue).

parseInt и parseFloat

Для явного преобразования к числу можно использовать `+` или `Number()`. Если строка не является в точности числом, то результат будет `NaN`:

```
alert( +"100px" ); // NaN
```

Единственное исключение — это пробелы в начале строки и в конце, они игнорируются.

В реальной жизни мы часто сталкиваемся со значениями у которых есть единица измерения, например `"100px"` или `"12pt"` в CSS. Также во множестве стран символ валюты записывается после номинала `"19€"`. Так как нам получить числовое значение из таких строк?

Для этого есть `parseInt` и `parseFloat`.

Они «читают» число из строки. Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число. Функция `parseInt` возвращает целое число, а `parseFloat` возвращает число с плавающей точкой:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12, вернётся только целая часть
alert( parseFloat('12.3.4') ); // 12.3, произойдёт остановка
чтения на второй точке
```

Функции `parseInt/parseFloat` вернут `NaN`, если не смогли прочитать ни одну цифру:

```
alert( parseInt('a123') ); // NaN, на первом символе
происходит остановка чтения
```

Второй аргумент `parseInt(str, radix)`

Функция `parseInt()` имеет необязательный второй параметр. Он определяет систему счисления, таким образом `parseInt` может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, без 0x тоже работает

alert( parseInt('2n9c', 36) ); // 123456
```

Другие математические функции

В JavaScript встроен объект Math, который содержит различные математические функции и константы.

Несколько примеров:

`Math.random()`

Возвращает псевдослучайное число в диапазоне от 0 (включительно) до 1 (но не включая 1)

```
alert( Math.random() ); // 0.1234567894322
alert( Math.random() ); // 0.5435252343232
alert( Math.random() ); // ... (любое количество псевдослучайных чисел)
```

`Math.max(a, b, c...)` / `Math.min(a, b, c...)`

Возвращает наибольшее/наименьшее число из перечисленных аргументов.

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5
alert( Math.min(1, 2) ); // 1
```

`Math.pow(n, power)`

Возвращает число `n`, возведённое в степень `power`

```
alert( Math.pow(2, 10) ); // 2 в степени 10 = 1024
```

В объекте `Math` есть множество функций и констант, включая тригонометрические функции, подробнее можно

ознакомиться в документации по объекту Math.

Итого

Чтобы писать числа с большим количеством нулей:

- Используйте краткую форму записи чисел – `"e"`, с указанным количеством нулей. Например: `123e6` это `123` с 6-ю нулями `123000000`.

  Отрицательное число после `"e"` приводит к делению числа на 1 с указанным количеством нулей. Например: `123e-6` это `0.000123` (`123` миллионных).

Для других систем счисления:

- Можно записывать числа сразу в шестнадцатеричной (`0x`), восьмеричной (`0o`) и бинарной (`0b`) системах счисления
- `parseInt(str, base)` преобразует строку в целое число в соответствии с указанной системой счисления: `2 ≤ base ≤ 36`.
- `num.toString(base)` представляет число в строковом виде в указанной системе счисления `base`.

Для преобразования значений типа `12pt` и `100px` в число:

- Используйте `parseInt/parseFloat` для «мягкого» преобразования строки в число, данные функции по порядку считывают число из строки до тех пор пока не возникнет ошибка.

Для дробей:

- Используйте округления `Math.floor`, `Math.ceil`, `Math.trunc`, `Math.round` или

`num.toFixed(precision)`.

Помните, что при работе с дробями происходит потеря точности.

Ещё больше математических функций:

Документация по объекту Math. Библиотека маленькая, но содержит всё самое важное.

Задачи

Сумма пользовательских чисел

важность: 5

Создайте скрипт, который запрашивает ввод двух чисел (используйте prompt) и после показывает их сумму.

Запустить демо

P.S. Есть «подводный камень» при работе с типами.

К решению

Почему 6.35.toFixed(1) == 6.3?

важность: 4

Методы `Math.round` и `toFixed`, согласно документации, округляют до ближайшего целого числа: `0..4` округляется в меньшую сторону, тогда как `5..9` в большую сторону.

Например:

```
alert( 1.35.toFixed(1) ); // 1.4
```

Но почему в примере ниже 6.35 округляется до 6.3?

```
alert( 6.35.toFixed(1) ); // 6.3
```

Как правильно округлить `6.35`?

К решению

Ввод числового значения

важность: 5

Создайте функцию `readNumber`, которая будет запрашивать ввод числового значения до тех пор, пока посетитель его не введёт.

Функция должна возвращать числовое значение.

Также надо разрешить пользователю остановить процесс ввода, отправив пустую строку или нажав «Отмена». В этом случае функция должна вернуть `null`.

Запустить демо

Открыть песочницу с тестами для задачи.

К решению

Бесконечный цикл по ошибке

важность: 4

Этот цикл – бесконечный. Он никогда не завершится,

почему?

```
let i = 0;
while (i != 10) {
  i += 0.2;
}
```

К решению

Случайное число от min до max

важность: 2

Встроенный метод `Math.random()` возвращает случайное число от `0` (включительно) до `1` (но не включая `1`)

Напишите функцию `random(min, max)`, которая генерирует случайное число с плавающей точкой от `min` до `max` (но не включая `max`).

Пример работы функции:

```
alert( random(1, 5) ); // 1.2345623452
alert( random(1, 5) ); // 3.7894332423
alert( random(1, 5) ); // 4.3435234525
```

К решению

Случайное целое число от min до max

важность: 2

Напишите функцию `randomInteger(min, max)`, которая генерирует случайное *целое* (integer) число от `min` до `max` (включительно).

Любое число из интервала `min..max` должно появляться с одинаковой вероятностью.

Пример работы функции:

```
alert( randomInteger(1, 5) ); // 1
alert( randomInteger(1, 5) ); // 3
alert( randomInteger(1, 5) ); // 5
```

Можно использовать решение из предыдущей задачи.

К решению

Строки

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков.

Внутренний формат для строк — всегда UTF-16, вне зависимости от кодировки страницы.

Кавычки

В JavaScript есть разные типы кавычек.

Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку мы сможем вставлять произвольные выражения, обернув их в `${…}`:

```
function sum(a, b) {
  return a + b;
}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Ещё одно преимущество обратных кавычек — они могут занимать более одной строки, вот так:

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;

alert(guestList); // список гостей, состоящий из нескольких
строк
```

Выглядит вполне естественно, не правда ли? Что тут такого? Но если попытаться использовать точно так же одинарные или двойные кавычки, то будет ошибка:

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
  * John";
```

Одинарные и двойные кавычки в языке с незапамятных времён: тогда потребность в многострочных строках не учитывалась. Что касается обратных кавычек, они появились существенно позже, и поэтому они гибче.

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: `func`string``. Автоматически вызываемая функция `func` получает строку и встроенные в неё выражения и может их обработать. Подробнее об

этом можно прочитать в документации. Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

Спецсимволы

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как `\n`:

```
let guestList = "Guests:\n * John\n * Pete\n * Mary";

alert(guestList); // список гостей, состоящий из нескольких строк
```

В частности, эти две строки эквивалентны, просто записаны по-разному:

```
// перевод строки добавлен с помощью символа перевода строки
let str1 = "Hello\nWorld";

// многострочная строка, созданная с использованием обратных кавычек
let str2 = `Hello
World`;

alert(str1 == str2); // true
```

Есть и другие, реже используемые спецсимволы. Вот список:

Символ	Описание
`\n`	Перевод строки
`\r`	Возврат каретки: самостоятельно не

	используется. В текстовых файлах Windows для перевода строки используется комбинация символов `\r\n`.
`\', \"`	Кавычки
`\\`	Обратный слеш
`\t`	Знак табуляции
`\b, \f, \v`	Backspace, Form Feed и Vertical Tab — оставлены для обратной совместимости, сейчас не используются.
`\xXX`	Символ с шестнадцатеричным юникодным кодом `XX`, например, `'\x7A'` — то же самое, что `'z'`.
`\uXXXX`	Символ в кодировке UTF-16 с шестнадцатеричным кодом `XXXX`, например, `\u00A9` — юникодное представление знака копирайта, ©. Код должен состоять ровно из 4 шестнадцатеричных цифр.
`\u{X...XXXXXX}` (от 1 до 6 шестнадцатеричных цифр)	Символ в кодировке UTF-32 с шестнадцатеричным кодом от U+0000 до U+10FFFF. Некоторые редкие символы

кодируются двумя 16-битными словами и занимают 4 байта. Так можно вставлять символы с длинным кодом.

Примеры с Юникодом:

```
// ©
alert( "\u00A9" );

// Длинные юникодные коды
// 佫, редкий китайский иероглиф
alert( "\u{20331}" );
// 😍, лицо с улыбкой и глазами в форме сердец
alert( "\u{1F60D}" );
```

Все спецсимволы начинаются с обратного слеша, `\` — так называемого «символа экранирования».

Он также используется, если необходимо вставить в строку кавычку.

К примеру:

```
alert( 'Im the Walrus!' ); //  the Walrus!
\'
I'm
```

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш — `\'` — иначе она бы обозначала окончание строки.

Разумеется, требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Так что мы можем применить и более элегантное решение, использовав для этой строки двойные или обратные кавычки:

```
alert( `I'm the Walrus!` ); // I'm the Walrus!
```

Заметим, что обратный слеш `\` служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ `\`. Вы можете явно видеть это в выводах `alert` в примерах выше.

Но что, если нам надо добавить в строку собственно сам обратный слеш `\`?

Это можно сделать, добавив перед ним… ещё один обратный слеш!

```
alert( `The backslash: \\` ); // The backslash: \
```

Длина строки

Свойство `length` содержит длину строки:

```
alert( `My\n`.length ); // 3
```

Обратите внимание, `\n` — это один спецсимвол, поэтому тут всё правильно: длина строки `3`.

`length` — это свойство

Бывает так, что люди с практикой в других языках случайно пытаются вызвать его, добавляя круглые скобки: они пишут `str.length()` вместо `str.length`. Это не работает.

Так как `str.length` — это числовое свойство, а не функция, добавлять скобки не нужно.

Доступ к символам

Получить символ, который занимает позицию `pos`, можно с помощью квадратных скобок: `[pos]`. Также можно использовать метод `charAt`: str.charAt(pos). Первый символ занимает нулевую позицию:

```
let str = `Hello`;

// получаем первый символ
alert( str[0] ); // H
alert( str.charAt(0) ); // H

// получаем последний символ
alert( str[str.length - 1] ); // o
```

Квадратные скобки — современный способ получить символ, в то время как `charAt` существует в основном по историческим причинам.

Разница только в том, что если символ с такой позицией отсутствует, тогда `[]` вернёт `undefined`, а `charAt` — пустую строку:

```
let str = `Hello`;

alert( str[1000] ); // undefined
alert( str.charAt(1000) ); // '' (пустая строка)
```

Также можно перебрать строку посимвольно, используя `for..of`:

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char — сначала "H", потом "e",
потом "l" и т. д.)
}
```

Строки неизменяемы

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана — она такая навсегда.

Давайте попробуем так сделать, и убедимся, что это не работает:

```
let str = 'Hi';

str[0] = 'h'; // ошибка
alert( str[0] ); // не работает
```

Можно создать новую строку и записать её в ту же самую переменную вместо старой.

Например:

```
let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi
```

В последующих разделах мы увидим больше примеров.

Изменение регистра

Методы toLowerCase() и toUpperCase() меняют регистр символов:

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Если мы захотим перевести в нижний регистр какой-то конкретный символ:

```
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

Поиск подстроки

Существует несколько способов поиска подстроки.

str.indexOf

Первый метод — str.indexOf(substr, pos).

Он ищет подстроку `substr` в строке `str`, начиная с позиции `pos`, и возвращает позицию, на которой располагается совпадение, либо `-1` при отсутствии совпадений.

Например:

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, потому что подстрока 'Widget' найдена в начале
alert( str.indexOf('widget') ); // -1, совпадений нет, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1, подстрока "id" найдена на позиции 1 (..idget with id)
```

Необязательный второй аргумент позволяет начать поиск с определённой позиции.

Например, первое вхождение `"id"` — на позиции `1`. Для того, чтобы найти следующее, начнём поиск с позиции `2`:

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

Чтобы найти все вхождения подстроки, нужно запустить `indexOf` в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:

```
let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска
```

```
let pos = 0;
while (true) {
  let foundPos = str.indexOf(target, pos);
  if (foundPos == -1) break;

  alert( `Найдено тут: ${foundPos}` );
  pos = foundPos + 1; // продолжаем со следующей позиции
}
```

Тот же алгоритм можно записать и короче:

```
let str = "Ослик Иа-Иа посмотрел на виадук";
let target = "Иа";

let pos = -1;
while ((pos = str.indexOf(target, pos + 1)) != -1) {
  alert( pos );
}
```

`str.lastIndexOf(substr, position)`

Также есть похожий метод str.lastIndexOf(substr, position), который ищет с конца строки к её началу.

Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

При проверке `indexOf` в условии `if` есть небольшое неудобство. Такое условие не будет работать:

```
let str = "Widget with id";

if (str.indexOf("Widget")) {
    alert("Совпадение есть"); // не работает
```

```
}
```

Мы ищем подстроку `"Widget"`, и она здесь есть, прямо на позиции `0`. Но `alert` не показывается, т. к. `str.indexOf("Widget")` возвращает `0`, и `if` решает, что тест не пройден.

Поэтому надо делать проверку на `-1`:

```
let str = "Widget with id";


    alert("Совпадение есть"); // теперь работает
}
if (str.indexOf("Widget") != -1) {
```

Трюк с побитовым НЕ

Существует старый трюк с использованием побитового оператора НЕ — `~`. Он преобразует число в 32-разрядное целое со знаком (signed 32-bit integer). Дробная часть, в случае, если она присутствует, отбрасывается. Затем все биты числа инвертируются.

На практике это означает простую вещь: для 32-разрядных целых чисел значение `~n` равно `-(n+1)`.

В частности:

```
alert( ~2 ); // -3, то же, что -(2+1)
alert( ~1 ); // -2, то же, что -(1+1)
alert( ~0 ); // -1, то же, что -(0+1)
alert( ~-1 ); // 0, то же, что -(-1+1)
```

Таким образом, `~n` равняется 0 только при `n == -1` (для любого `n`, входящего в 32-разрядные целые числа со знаком).

Соответственно, прохождение проверки `if ( ~str.indexOf("…") )` означает, что результат `indexOf` отличен от `-1`, совпадение есть.

Это иногда применяют, чтобы сделать проверку `indexOf` компактнее:

```
let str = "Widget";

if (~str.indexOf("Widget")) {
  alert( 'Совпадение есть' ); // работает
}
```

Обычно использовать возможности языка каким-либо неочевидным образом не рекомендуется, но этот трюк широко используется в старом коде, поэтому его важно понимать.

Просто запомните: `if (~str.indexOf(…))` означает «если найдено».

Впрочем, если быть точнее, из-за того, что большие числа обрезаются до 32 битов оператором `~`, существуют другие числа, для которых результат тоже будет `0`, самое маленькое из которых — `~4294967295=0`. Поэтому такая проверка будет правильно работать только для строк меньшей длины.

На данный момент такой трюк можно встретить только в старом коде, потому что в новом он просто не нужен: есть метод `.includes` (см. ниже).

includes, startsWith, endsWith

Более современный метод str.includes(substr, pos) возвращает `true`, если в строке `str` есть подстрока `substr`, либо `false`, если нет.

Это — правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна:

```
alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false
```

Необязательный второй аргумент `str.includes` позволяет начать поиск с определённой позиции:

```
alert( "Midget".includes("id") ); // true
alert( "Midget".includes("id", 3) ); // false, поиск начат с
позиции 3
```

Методы str.startsWith и str.endsWith проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

```
alert( "Widget".startsWith("Wid") ); // true, "Wid" — начало
"Widget"
alert( "Widget".endsWith("get") ); // true, "get" — окончание
"Widget"
```

Получение подстроки

В JavaScript есть 3 метода для получения подстроки: `substring`, `substr` и `slice`.

`str.slice(start [, end])`

Возвращает часть строки от `start` до (не включая) `end`.

Например:

```
let str = "stringify";
// 'strin', символы от 0 до 5 (не включая 5)
alert( str.slice(0, 5) );
// 's', от 0 до 1, не включая 1, т. е. только один символ на
позиции 0
alert( str.slice(0, 1) );
```

Если аргумент `end` отсутствует, `slice` возвращает символы до конца строки:

```
let str = "st";
alert( str.slice(2) ); // ringify, с позиции 2 и до конца
```

```
ringify
```

Также для `start/end` можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов *с конца строки*:

```
let str = "striny";

// начинаем с позиции 4 справа, а заканчиваем на позиции 1 справа
alert( str.slice(-4, -1) ); // gif
```
```
gif
```

`str.substring(start [, end])`

Возвращает часть строки *между* `start` и `end`.

Это — почти то же, что и `slice`, но можно задавать `start` больше `end`.

Например:

```
let str = "stify";

// для substring эти два примера — одинаковы
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// …но не для slice:
alert( str.slice(2, 6) ); // "ring" (то же самое)
alert( str.slice(6, 2) ); // "" (пустая строка)
```
```
ring
```

Отрицательные значения `substring`, в отличие от `slice`, не поддерживает, они интерпретируются как `0`.

`str.substr(start [, length])`

Возвращает часть строки от `start` длины `length`.

В противоположность предыдущим методам, этот

позволяет указать длину вместо конечной позиции:

```
let str = "stify";
// ring, получаем 4 символа, начиная с позиции 2
alert( str.substr(2, 4) );
```
```
ring
```

Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:

```
let str = "strinfy";
// gi, получаем 2 символа, начиная с позиции 4 с конца строки
alert( str.substr(-4, 2) );
```
```
gi
```

Давайте подытожим, как работают эти методы, чтобы не запутаться:

метод	выбирает...	отрицательные значения
`slice(start, end)`	от `start` до `end` (не включая `end`)	можно передавать отрицательные значения
`substring(start, end)`	между `start` и `end`	отрицательные значения равнозначны `0`
`substr(start, length)`	`length` символов, начиная от `start`	значение `start` может быть отрицательным

Какой метод выбрать?

Все эти методы эффективно выполняют задачу. Формально у метода `substr` есть небольшой недостаток: он описан не в собственно спецификации JavaScript, а в приложении к ней — Annex B. Это приложение описывает возможности языка для использования в браузерах, существующие в основном по историческим причинам. Таким образом, в другом окружении, отличном от браузера, он может не поддерживаться. Однако на практике он работает везде.

Из двух других вариантов, `slice` более гибок, он поддерживает отрицательные аргументы, и его короче писать. Так что, в принципе, можно запомнить только его.

Сравнение строк

Как мы знаем из главы Операторы сравнения, строки сравниваются посимвольно в алфавитном порядке.

Тем не менее, есть некоторые нюансы.

1. Строчные буквы больше заглавных:

```
alert( 'a' > 'Z' ); // true
```

2. Буквы, имеющие диакритические знаки, идут «не по порядку»:

```
alert( 'Österreich' > 'Zealand' ); // true
```

Это может привести к своеобразным результатам при сортировке названий стран: нормально было бы ожидать, что `Zealand` будет после `Österreich` в списке.

Чтобы разобраться, что происходит, давайте ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в UTF-16. Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот.

`str.codePointAt(pos)`

Возвращает код для символа, находящегося на позиции `pos`:

```
// одна и та же буква в нижнем и верхнем регистре
// будет иметь разные коды
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

`String.fromCodePoint(code)`

Создаёт символ по его коду `code`

```
alert( String.fromCodePoint(90) ); // Z
```

Также можно добавлять юникодные символы по их кодам, используя `\u` с шестнадцатеричным кодом символа:

```
// 90 – 5a в шестнадцатеричной системе счисления
alert( '\u005a' ); // Z
```

Давайте сделаем строку, содержащую символы с кодами от `65` до `220` — это латиница и ещё некоторые распространённые символы:

```
let str = '';

for (let i = 65; i <= 220; i++) {
  str += String.fromCodePoint(i);
}
alert( str );
//
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
~□□□□□
// ¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ
```

Как видите, сначала идут заглавные буквы, затем несколько спецсимволов, затем строчные и `Ö` ближе к концу вывода.

Теперь очевидно, почему `a > Z`.

Символы сравниваются по их кодам. Больший код — больший символ. Код `a` (97) больше кода `Z` (90).

> Все строчные буквы идут после заглавных, так как их коды больше.
>
> Некоторые буквы, такие как `Ö`, вообще находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от `a` до `z`.

Правильное сравнение

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты.

Поэтому браузеру нужно знать, какой язык использовать для сравнения.

К счастью, все современные браузеры (для IE10– нужна дополнительная библиотека Intl.JS) поддерживают стандарт ECMA 402, обеспечивающий правильное сравнение строк на разных языках с учётом их правил.

Для этого есть соответствующий метод.

Вызов str.localeCompare(str2) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

> Отрицательное число, если `str` меньше `str2`.
>
> Положительное число, если `str` больше `str2`.

`0`, если строки равны.

Например:

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```

У этого метода есть два дополнительных аргумента, которые указаны в документации. Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между `"a"` и `"á"`.

Как всё устроено, Юникод

Глубокое погружение в тему

Этот раздел более подробно описывает, как устроены строки. Такие знания пригодятся, если вы намерены работать с эмодзи, редкими математическими символами, иероглифами, либо с ещё какими-то редкими символами.

Если вы не планируете их поддерживать, эту секцию можно пропустить.

Суррогатные пары

Многие символы возможно записать одним 16-битным словом: это и буквы большинства европейских языков, и числа, и даже многие иероглифы.

Но 16 битов — это 65536 комбинаций, так что на все символы этого, разумеется, не хватит. Поэтому редкие символы записываются двумя 16-битными словами — это также называется «суррогатная пара».

Длина таких строк — `2`:

```
alert( '𝒳'.length ); // 2, MATHEMATICAL SCRIPT CAPITAL X
alert( '😂'.length ); // 2, FACE WITH TEARS OF JOY
alert( '𩷶'.length ); // 2, редкий китайский иероглиф
```

Обратите внимание, суррогатные пары не существовали, когда был создан JavaScript, поэтому язык не обрабатывает их адекватно!

Ведь в каждой из этих строк только один символ, а `length` показывает длину `2`.

`String.fromCodePoint` и `str.codePointAt` — два редких метода, правильно работающие с суррогатными парами, но они и появились в языке недавно. До них были только String.fromCharCode и str.charCodeAt. Эти методы, вообще, делают то же самое, что `fromCodePoint/codePointAt`, но не работают с суррогатными парами.

Получить символ, представленный суррогатной парой, может быть не так просто, потому что суррогатная пара интерпретируется как два символа:

```
alert( '𝒳'[0] ); // странные символы…
alert( '𝒳'[1] ); // …части суррогатной пары
```

Части суррогатной пары не имеют смысла сами по себе, так что вызовы `alert` в этом примере покажут лишь

мусор.

Технически, суррогатные пары возможно обнаружить по их кодам: если код символа находится в диапазоне `0xd800..0xdbff`, то это — первая часть суррогатной пары. Следующий символ — вторая часть — имеет код в диапазоне `0xdc00..0xdfff`. Эти два диапазона выделены исключительно для суррогатных пар по стандарту.

В данном случае:

```
// charCodeAt не поддерживает суррогатные пары, поэтому возвращает код для их частей

alert( '𝒳'.charCodeAt(0).toString(16) ); // d835, между 0xd800 и 0xdbff
alert( '𝒳'.charCodeAt(1).toString(16) ); // dcb3, между 0xdc00 и 0xdfff
```

Дальше в главе Перебираемые объекты будут ещё способы работы с суррогатными парами. Для этого есть и специальные библиотеки, но нет достаточно широко известной, чтобы предложить её здесь.

Диакритические знаки и нормализация

Во многих языках есть символы, состоящие из некоторого основного символа со знаком сверху или снизу.

Например, буква `a` — это основа для `àáâäãåā`. Наиболее используемые составные символы имеют свой собственный код в таблице UTF-16. Но не все, в силу большого количества комбинаций.

Чтобы поддерживать любые комбинации, UTF-16 позволяет использовать несколько юникодных символов: основной и дальше один или несколько особых символов-знаков.

Например, если после `S` добавить специальный символ «точка сверху» (код `\u0307`), отобразится S̈.

```
alert( 'S\u0307' ); // Ṡ
```

Если надо добавить сверху (или снизу) ещё один знак — без проблем, просто добавляем соответствующий символ.

Например, если добавить символ «точка снизу» (код `\u0323`), отобразится S с точками сверху и снизу: Ṩ.

Добавляем два символа:

```
alert( 'S\u0307\u0323' ); // Ṩ
```

Это даёт большую гибкость, но из-за того, что порядок дополнительных символов может быть различным, мы получаем проблему сравнения символов: можно представить по-разному символы, которые ничем визуально не отличаются.

Например:

```
let s1 = 'S\u0307\u0323'; // Ṩ, S + точка сверху + точка снизу
let s2 = 'S\u0323\u0307'; // Ṩ, S + точка снизу + точка сверху

alert( `s1: ${s1}, s2: ${s2}` );

alert( s1 == s2 ); // false, хотя на вид символы одинаковы
(?!)
```

Для решения этой проблемы есть алгоритм «юникодной нормализации», приводящий каждую строку к единому «нормальному» виду.

Его реализует метод str.normalize().

```
alert( "S\u0307\u0323".normalize() == "S
\u0323\u0307".normalize() ); // true
```

Забавно, но в нашем случае `normalize()` «схлопывает» последовательность из трёх символов в один: `\u1e68` —

S с двумя точками.

```
alert( "S\u0307\u0323".normalize().length ); // 1

alert( "S\u0307\u0323".normalize() == "\u1e68" ); // true
```

Разумеется, так происходит не всегда. Просто Ṩ — это достаточно часто используемый символ, поэтому создатели UTF-16 включили его в основную таблицу и присвоили ему код.

Подробнее о правилах нормализации и составлении символов можно прочитать в дополнении к стандарту Юникод: Unicode Normalization Forms. Для большинства практических целей информации из этого раздела достаточно.

Итого

- Есть три типа кавычек. Строки, использующие обратные кавычки, могут занимать более одной строки в коде и включать выражения `${…}`.
- Строки в JavaScript кодируются в UTF-16.
- Есть специальные символы, такие как `\n`, и можно добавить символ по его юникодному коду, используя `\u…`.
- Для получения символа используйте `[]`.
- Для получения подстроки используйте `slice` или `substring`.
- Для того, чтобы перевести строку в нижний или верхний регистр, используйте `toLowerCase/toUpperCase`.
- Для поиска подстроки используйте `indexOf` или `includes/startsWith/endsWith`, когда надо только проверить, есть ли вхождение.
- Чтобы сравнить строки с учётом правил языка,

используйте `localeCompare`.

Строки также имеют ещё кое-какие полезные методы:

- `str.trim()` — убирает пробелы в начале и конце строки.
- `str.repeat(n)` — повторяет строку `n` раз.
- …и другие, которые вы можете найти в справочнике.

Также есть методы для поиска и замены с использованием регулярных выражений. Но это отдельная большая тема, поэтому ей посвящена отдельная глава учебника Регулярные выражения.

Задачи

Сделать первый символ заглавным

важность: 5

Напишите функцию `ucFirst(str)`, возвращающую строку `str` с заглавным первым символом. Например:

```
ucFirst("вася") == "Вася";
```

Открыть песочницу с тестами для задачи.

К решению

Проверка на спам

важность: 5

Напишите функцию `checkSpam(str)`, возвращающую `true`, если `str` содержит `'viagra'` или `'XXX'`, а иначе `false`.

Функция должна быть нечувствительна к регистру:

```
checkSpam('buy ViAgRA now') == true
checkSpam('free xxxxx') == true
checkSpam("innocent rabbit") == false
```

Открыть песочницу с тестами для задачи.

К решению

Усечение строки

важность: 5

Создайте функцию `truncate(str, maxlength)`, которая проверяет длину строки `str` и, если она превосходит `maxlength`, заменяет конец `str` на `"…"`, так, чтобы её длина стала равна `maxlength`.

Результатом функции должна быть та же строка, если усечение не требуется, либо, если необходимо, усечённая строка.

Например:

```
truncate("Вот, что мне хотелось бы сказать на эту тему:", 20)
= "Вот, что мне хотело…"

truncate("Всем привет!", 20) = "Всем привет!"
```

Открыть песочницу с тестами для задачи.

К решению

Выделить число

важность: 4

Есть стоимость в виде строки `"$120"`. То есть сначала идёт знак валюты, а затем – число.

Создайте функцию `extractCurrencyValue(str)`, которая будет из такой строки выделять числовое значение и возвращать его.

Например:

```
alert( extractCurrencyValue('$120') === 120 ); // true
```

Открыть песочницу с тестами для задачи.

К решению

Массивы

Объекты позволяют хранить данные со строковыми ключами. Это замечательно.

Но довольно часто мы понимаем, что нам необходима *упорядоченная коллекция* данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, `Array`.

Объявление

Существует два варианта синтаксиса для создания пустого массива:

```
let arr = new Array();
let arr = [];
```

Практически всегда используется второй вариант синтаксиса. В скобках мы можем указать начальные значения элементов:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];
```

Элементы массива нумеруются, начиная с нуля.

Мы можем получить элемент, указав его номер в квадратных скобках:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко
alert( fruits[1] ); // Апельсин
alert( fruits[2] ); // Слива
```

Мы можем заменить элемент:

```
fruits[2] = 'Груша'; // теперь ["Яблоко", "Апельсин", "Груша"]
```

…Или добавить новый к существующему массиву:

```
fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша",
"Лимон"]
```

Общее число элементов массива содержится в его свойстве `length`:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits.length ); // 3
```

Вывести массив целиком можно при помощи `alert`.

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits ); // Яблоко, Апельсин, Слива
```

В массиве могут храниться элементы любого типа.

Например:

```
// разные типы значений
let arr = [ 'Яблоко', { name: 'Джон' }, true, function() {
alert('привет'); } ];

// получить элемент с индексом 1 (объект) и затем показать его
свойство
alert( arr[1].name ); // Джон

// получить элемент с индексом 3 (функция) и выполнить её
arr[3](); // привет
```

Висячая запятая

Список элементов массива, как и список свойств

объекта, может оканчиваться запятой:

```
let fruits = [
  "Яблоко",
  "Апельсин",
  "Слива"
];
,
```

«Висячая запятая» упрощает процесс добавления/удаления элементов, так как все строки становятся идентичными.

Методы pop/push, shift/unshift

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

`push` добавляет элемент в конец.

`shift` удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

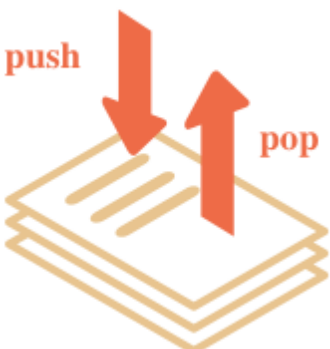
Существует и другой вариант применения для массивов – структура данных, называемая стек.

Она поддерживает два вида операций:

- `push` добавляет элемент в конец.
- `pop` удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца».

Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху:



Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.

В компьютерных науках структура данных, делающая это возможным, называется двусторонняя очередь.

Методы, работающие с концом массива:

`pop`

Удаляет последний элемент из массива и возвращает его:

```
let fruits = ["Яблоко", "Апельсин", "Груша"];
```

```
alert( fruits.pop() ); // удаляем "Груша" и выводим его

alert( fruits ); // Яблоко, Апельсин
```

push

Добавляет элемент в конец массива:

```
let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша
```

Вызов `fruits.push(...)` равнозначен `fruits[fruits.length] = ...`.

Методы, работающие с началом массива:

shift

Удаляет из массива первый элемент и возвращает его:

```
let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.shift() ); // удаляем Яблоко и выводим его

alert( fruits ); // Апельсин, Груша
```

unshift

Добавляет элемент в начало массива:

```
let fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша
```

Методы `push` и `unshift` могут добавлять сразу несколько элементов:

```
let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");
fruits.unshift("Ананас", "Лимон");

// ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]
alert( fruits );
```

Внутреннее устройство массива

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству `arr[0]` – это по сути обычный синтаксис доступа по ключу, как `obj[key]`, где в роли `obj` у нас `arr`, а в качестве ключа – числовой индекс.

Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство `length`. Но в основе всё равно лежит объект.

Следует помнить, что в JavaScript существует 8 основных типов данных. Массив является объектом и, следовательно, ведёт себя как объект.

Например, копируется по ссылке:

```
let fruits = ["Банан"]

let arr = fruits; // копируется по ссылке (две переменные ссылаются на один и тот же массив)

alert( arr === fruits ); // true

arr.push("Груша"); // массив меняется по ссылке

alert( fruits ); // Банан, Груша – теперь два элемента
```

…Но то, что действительно делает массивы особенными – это их внутреннее представление. Движок JavaScript старается хранить элементы массива в непрерывной

области памяти, один за другим, так, как это показано на иллюстрациях к этой главе. Существуют и другие способы оптимизации, благодаря которым массивы работают очень быстро.

Но все они утратят эффективность, если мы перестанем работать с массивом как с «упорядоченной коллекцией данных» и начнём использовать его как обычный объект.

Например, технически мы можем сделать следующее:

```
let fruits = []; // создаём массив

fruits[99999] = 5; // создаём свойство с индексом, намного
превышающим длину массива

fruits.age = 25; // создаём свойство с произвольным именем
```

Это возможно, потому что в основе массива лежит объект. Мы можем присвоить ему любые свойства.

Но движок поймёт, что мы работаем с массивом, как с обычным объектом. Способы оптимизации, используемые для массивов, в этом случае не подходят, поэтому они будут отключены и никакой выгоды не принесут.

Варианты неправильного применения массива:

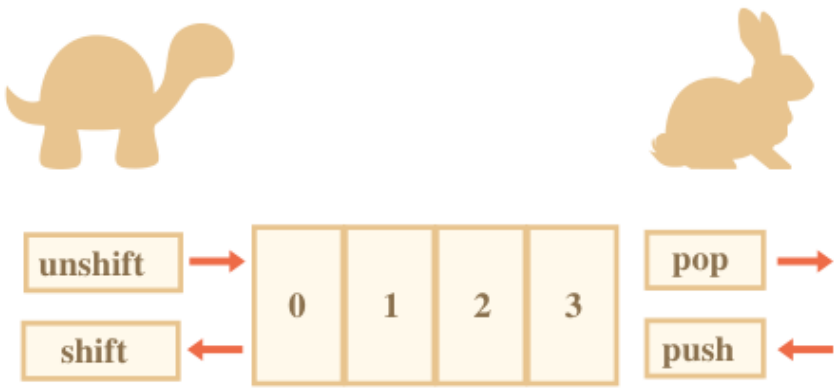
- Добавление нечислового свойства, например: `arr.test = 5`.
- Создание «дыр», например: добавление `arr[0]`, затем `arr[1000]` (между ними ничего нет).
- Заполнение массива в обратном порядке, например: `arr[1000]`, `arr[999]` и т.д.

Массив следует считать особой структурой, позволяющей работать с *упорядоченными данными*. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, пожалуйста,

используйте их именно в таких случаях. Если вам нужны произвольные ключи, вполне возможно, лучше подойдёт обычный объект `{}`.

Эффективность

Методы `push/pop` выполняются быстро, а методы `shift/unshift` – медленно.



Почему работать с концом массива быстрее, чем с его началом? Давайте посмотрим, что происходит во время выполнения:

```
fruits.shift(); // удаляем первый элемент с начала
```

Просто взять и удалить элемент с номером `0` недостаточно. Нужно также заново пронумеровать остальные элементы.

Операция `shift` должна выполнить 3 действия:

1. Удалить элемент с индексом `0`.
2. Сдвинуть все элементы влево, заново пронумеровать их, заменив `1` на `0`, `2` на `1` и т.д.

3. Обновить свойство `length`.

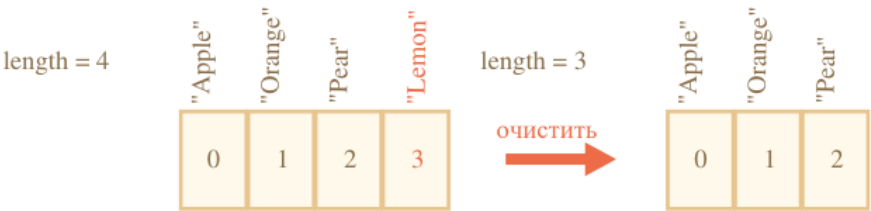
Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.

То же самое происходит с `unshift`: чтобы добавить элемент в начало массива, нам нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

А что же с `push/pop`? Им не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод `pop` очищает индекс и уменьшает значение `length`.

Действия при операции `pop`:

```
fruits.pop(); // удаляем один элемент с конца
```



Метод `pop` не требует перемещения, потому что остальные элементы остаются с теми же индексами. Именно поэтому он выполняется очень быстро.

Аналогично работает метод `push`.

Перебор элементов

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам:

```
let arr = ["Яблоко", "Апельсин", "Груша"];


  alert( arr[i] );
}
for (let i = 0; i < arr.length; i++) {
```

Но для массивов возможен и другой вариант цикла, `for..of`:

```
let fruits = ["Яблоко", "Апельсин", "Слива"];

// проходит по значениям
for (let fruit of fruits) {
  alert( fruit );
}
```

Цикл `for..of` не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

Технически, так как массив является объектом, можно использовать и вариант `for..in`:

```
let arr = ["Яблоко", "Апельсин", "Груша"];


  alert( arr[key] ); // Яблоко, Апельсин, Груша
}
for (let key in arr) {
```

Но на самом деле это – плохая идея. Существуют скрытые недостатки этого способа:

1. Цикл `for..in` выполняет перебор *всех свойств* объекта, а не только цифровых.

   В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые *выглядят, как массив*. То есть, у них есть свойство `length` и индексы, но они также могут иметь дополнительные нечисловые свойства и методы, которые нам обычно не нужны. Тем не менее, цикл `for..in` выведет и их. Поэтому, если нам приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

2. Цикл `for..in` оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее. Увеличение скорости выполнения может иметь значение только при возникновении узких мест. Но мы всё же должны представлять разницу.

В общем, не следует использовать цикл `for..in` для массивов.

Немного о «length»

Свойство `length` автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а наибольший цифровой индекс плюс один.

Например, единственный элемент, имеющий большой индекс, даёт большую длину:

```
let fruits = [];
fruits[123] = "Яблоко";

alert( fruits.length ); // 124
```

Обратите внимание, что обычно мы не используем массивы таким образом.

Ещё один интересный факт о свойстве `length` – его можно перезаписать.

Если мы вручную увеличим его, ничего интересного не произойдёт. Зато, если мы уменьшим его, массив станет короче. Этот процесс необратим, как мы можем понять из примера:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // укорачиваем до двух элементов
alert( arr ); // [1, 2]

arr.length = 5; // возвращаем length как было
alert( arr[3] ); // undefined: значения не восстановились
```

Таким образом, самый простой способ очистить массив – это `arr.length = 0;`.

new Array()

Существует ещё один вариант синтаксиса для создания массива:

```
let arr = ("Яблоко", "Груша", "и тд");
new
 Array
```

Он редко применяется, так как квадратные скобки `[]` короче. Кроме того, у него есть хитрая особенность.

Если `new Array` вызывается с одним аргументом, который представляет собой число, он создаёт массив *без элементов, но с заданной длиной*.

Давайте посмотрим, как можно оказать себе медвежью услугу:

```
let arr = new Array(2); // создастся ли массив [2]?

alert( arr[0] ); // undefined! нет элементов.

alert( arr.length ); // length 2
```

Как мы видим, в коде, представленном выше, в `new Array(number)` все элементы равны `undefined`.

Чтобы избежать появления таких неожиданных ситуаций, мы обычно используем квадратные скобки, если, конечно, не знаем точно, что по какой-то причине нужен именно `Array`.

Многомерные массивы

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

alert( matrix[1][1] ); // 5, центральный элемент
```

toString

Массивы по-своему реализуют метод `toString`, который возвращает список элементов, разделённых запятыми.

Например:

```
let arr = [1, 2, 3];

alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Давайте теперь попробуем следующее:

```
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
```

Массивы не имеют ни `Symbol.toPrimitive`, ни функционирующего `valueOf`, они реализуют только преобразование `toString`, таким образом, здесь `[]` становится пустой строкой, `[1]` становится `"1"`, а `[1,2]` становится `"1,2"`.

Когда бинарный оператор плюс `"+"` добавляет что-либо к строке, он тоже преобразует это в строку, таким образом:

```
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

Итого

Массив – это особый тип объекта, предназначенный для работы с упорядоченным набором элементов.

Объявление:

```
// квадратные скобки (обычно)
let arr = [item1, item2...];

// new Array (очень редко)
let arr = new Array(item1, item2...);
```

Вызов `new Array(number)` создаёт массив с заданной длиной, но без элементов.

Свойство `length` отражает длину массива или, если точнее, его последний цифровой индекс плюс один. Длина корректируется автоматически методами массива.

Если мы уменьшаем `length` вручную, массив укорачивается.

Мы можем использовать массив как двустороннюю очередь, используя следующие операции:

`push(...items)` добавляет `items` в конец массива.

`pop()` удаляет элемент в конце массива и возвращает его.

`shift()` удаляет элемент в начале массива и возвращает его.

`unshift(...items)` добавляет `items` в начало массива.

Чтобы пройтись по элементам массива:

`for (let i=0; i<arr.length; i++)` – работает быстрее всего, совместим со старыми браузерами.

`for (let item of arr)` – современный синтаксис только для значений элементов (к индексам нет доступа).

`for (let i in arr)` – никогда не используйте для массивов!

Мы вернёмся к массивам и изучим другие методы добавления, удаления, выделения элементов и сортировки массивов в главе: Методы массивов.

Задачи

Скопирован ли массив?

важность: 3

Что выведет следующий код?

```
let fruits = ["Яблоки", "Груша", "Апельсин"];

// добавляем новое значение в "копию"
let shoppingCart = fruits;
shoppingCart.push("Банан");

// что в fruits?
alert( fruits.length ); // ?
```

К решению

Операции с массивами

важность: 5

Давайте произведём 5 операций с массивом.

1. Создайте массив `styles` с элементами «Джаз» и «Блюз».
2. Добавьте «Рок-н-ролл» в конец.
3. Замените значение в середине на «Классика». Ваш код для поиска значения в середине должен работать для массивов с любой длиной.
4. Удалите первый элемент массива и покажите его.
5. Вставьте «Рэп» и «Регги» в начало массива.

Массив по ходу выполнения операций:

```
Джаз, Блюз
Джаз, Блюз, Рок-н-ролл
Джаз, Классика, Рок-н-ролл
Классика, Рок-н-ролл
```

```
Рэп, Регги, Классика, Рок-н-ролл
```

К решению

Вызов в контексте массива

важность: 5

Каков результат? Почему?

```
let arr = ["a", "b"];

arr.push(function() {
  alert( this );
})

arr[2](); // ?
```

К решению

Сумма введённых чисел

важность: 4

Напишите функцию `sumInput()`, которая:

- Просит пользователя ввести значения, используя `prompt` и сохраняет их в массив.
- Заканчивает запрашивать значения, когда пользователь введёт не числовое значение, пустую строку или нажмёт «Отмена».
- Подсчитывает и возвращает сумму элементов массива.

P.S. Ноль `0` – считается числом, не останавливайте ввод значений при вводе «0».

Запустить демо

К решению

Подмассив наибольшей суммы

важность: 2

На входе массив чисел, например: `arr = [1, -2, 3, 4, -9, 6]`.

Задача: найти непрерывный подмассив в `arr`, сумма элементов в котором максимальна.

Функция `getMaxSubSum(arr)` должна возвращать эту сумму.

Например:

```
getMaxSubSum([-1, , -9]) = 5 (сумма выделенных)
getMaxSubSum([, -9]) = 6
getMaxSubSum([-1, 2, 3, -9, ]) = 11
getMaxSubSum([-2, -1, ]) = 3
getMaxSubSum([, -9, 2, -3, 5]) = 100
getMaxSubSum([]) = 6 (берём все)
2
, 3
2
, -1, 2, 3
11
1
, 2
100
1
, 2, 3
```

Если все элементы отрицательные – ничего не

берём(подмассив пустой) и сумма равна «0»:

```
getMaxSubSum([-1, -2, -3]) = 0
```

Попробуйте придумать быстрое решение: $O(n^2)$, а лучше за O(n) операций.

Открыть песочницу с тестами для задачи.

К решению

Методы массивов

Массивы предоставляют множество методов. Чтобы было проще, в этой главе они разбиты на группы.

Добавление/удаление элементов

Мы уже знаем методы, которые добавляют и удаляют элементы из начала или конца:

- `arr.push(...items)` – добавляет элементы в конец,
- `arr.pop()` – извлекает элемент из конца,
- `arr.shift()` – извлекает элемент из начала,
- `arr.unshift(...items)` – добавляет элементы в начало.

Есть и другие.

splice

Как удалить элемент из массива?

Так как массивы – это объекты, то можно попробовать `delete`:

```
let arr = ["I", "go", "home"];

delete arr[1]; // удалить "go"

alert( arr[1] ); // undefined

// теперь arr = ["I",  , "home"];
alert( arr.length ); // 3
```

Вроде бы, элемент и был удалён, но при проверке оказывается, что массив всё ещё имеет 3 элемента `arr.length == 3`.

Это нормально, потому что всё, что делает `delete obj.key` – это удаляет значение с данным ключом `key`. Это нормально для объектов, но для массивов мы обычно хотим, чтобы оставшиеся элементы сдвинулись и заняли освободившееся место. Мы ждём, что массив станет короче.

Поэтому для этого нужно использовать специальные методы.

Метод arr.splice(str) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

Он начинает с позиции `index`, удаляет `deleteCount` элементов и вставляет `elem1, ..., elemN` на их место. Возвращает массив из удалённых элементов.

Этот метод проще всего понять, рассмотрев примеры.

Начнём с удаления:

```
let arr = ["Я", "изучаю", "JavaScript"];


alert( arr ); // осталось ["Я", "JavaScript"]
arr.splice(1, 1); // начиная с позиции 1,
удалить 1 элемент
```

Легко, правда? Начиная с позиции `1`, он убрал `1` элемент.

В следующем примере мы удалим 3 элемента и заменим их двумя другими.

```
let arr = [ "прямо", "сейчас"];

// удалить 3 первых элемента и заменить их другими
arr.splice(0, 3, "Давай", "танцевать");

alert( arr ) // теперь [, "прямо", "сейчас"]
"Я"
, "изучаю", "JavaScript",
"Давай", "танцевать"
```

Здесь видно, что `splice` возвращает массив из удалённых элементов:

```
let arr = [ "JavaScript", "прямо", "сейчас"];

// удалить 2 первых элемента
let removed = arr.splice(0, 2);

alert( removed ); // "Я", "изучаю" <-- массив из удалённых
элементов
"Я"
, "изучаю",
```

Метод `splice` также может вставлять элементы без удаления, для этого достаточно установить `deleteCount` в `0`:

```
let arr = ["Я", "изучаю", "JavaScript"];

// с позиции 2
// удалить 0 элементов
// вставить "сложный", "язык"
arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык",
"JavaScript"
```

Отрицательные индексы разрешены

В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца, как тут:

```
let arr = [1, 2, 5];

// начиная с индекса -1 (перед последним элементом)
// удалить 0 элементов,
// затем вставить числа 3 и 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

slice

Метод arr.slice намного проще, чем похожий на него `arr.splice`.

Его синтаксис:

```
arr.slice([start], [end])
```

Он возвращает новый массив, в который копирует элементы, начиная с индекса `start` и до `end` (не включая `end`). Оба индекса `start` и `end` могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива.

Это похоже на строковый метод `str.slice`, но вместо подстрок возвращает подмассивы.

Например:

```
let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s (копирует с 1 до 3)

alert( arr.slice(-2) ); // s,t (копирует с -2 до конца)
```

Можно вызвать `slice` и вообще без аргументов: `arr.slice()` создаёт копию массива `arr`. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

concat

Метод arr.concat создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

Его синтаксис:

```
arr.concat(arg1, arg2...)
```

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями.

В результате мы получаем новый массив, включающий в себя элементы из `arr`, а также `arg1`, `arg2` и так далее…

Если аргумент `argN` – массив, то все его элементы копируются. Иначе скопируется сам аргумент.

Например:

```
let arr = [1, 2];

// создать массив из: arr и [3,4]
```

```
alert( arr.concat([3, 4]) ); // 1,2,3,4

// создать массив из: arr и [3,4] и [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// создать массив из: arr и [3,4], потом добавить значения 5 и
6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Обычно он просто копирует элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть:

```
let arr = [1, 2];

let arrayLike = {
  0: "что-то",
  length: 1
};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

…Но если объект имеет специальное свойство `Symbol.isConcatSpreadable`, то он обрабатывается `concat` как массив: вместо него добавляются его числовые свойства.

Для корректной обработки в объекте должны быть числовые свойства и `length`:

```
let arr = [1, 2];

let arrayLike = {
  0: "что-то",
  1: "ещё",

  length: 2
};

alert( arr.concat(arrayLike) ); // 1,2,что-то,ещё
  [Symbol.isConcatSpreadable]: true,
```

Перебор: forEach

Метод arr.forEach позволяет запускать функцию для каждого элемента массива.

Его синтаксис:

```
arr.forEach(function(item, index, array) {
  // ... делать что-то с item
});
```

Например, этот код выведет на экран каждый элемент массива:

```
// Вызов alert для каждого элемента
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

А этот вдобавок расскажет и о своей позиции в массиве:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) =>
{
  alert(`${item} имеет позицию ${index} в ${array}`);
});
```

Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

Поиск в массиве

Далее рассмотрим методы, которые помогут найти что-нибудь в массиве.

indexOf/lastIndexOf и includes

Методы arr.indexOf, arr.lastIndexOf и arr.includes имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

`arr.indexOf(item, from)` ищет `item`, начиная с

индекса `from`, и возвращает индекс, на котором был найден искомый элемент, в противном случае `-1`.

`arr.lastIndexOf(item, from)` – то же самое, но ищет справа налево.

`arr.includes(item, from)` – ищет `item`, начиная с индекса `from`, и возвращает `true`, если поиск успешен.

Например:

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true
```

Обратите внимание, что методы используют строгое сравнение `===`. Таким образом, если мы ищем `false`, он находит именно `false`, а не ноль.

Если мы хотим проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является `arr.includes`.

Кроме того, очень незначительным отличием `includes` является то, что он правильно обрабатывает `NaN` в отличие от `indexOf/lastIndexOf`:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (должен быть 0, но ===
проверка на равенство не работает для NaN)
alert( arr.includes(NaN) );// true (верно)
```

find и findIndex

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод arr.find.

Его синтаксис таков:

```
let result = arr.find(function(item, index, array) {
  // если true – возвращается текущий элемент и перебор
прерывается
  // если все итерации оказались ложными, возвращается
undefined
});
```

Функция вызывается по очереди для каждого элемента массива:

- `item` – очередной элемент.
- `index` – его индекс.
- `array` – сам массив.

Если функция возвращает `true`, поиск прерывается и возвращается `item`. Если ничего не найдено, возвращается `undefined`.

Например, у нас есть массив пользователей, каждый из которых имеет поля `id` и `name`. Попробуем найти того, кто с `id == 1`:

```
let users = [
  {id: 1, name: "Вася"},
  {id: 2, name: "Петя"},
  {id: 3, name: "Маша"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // Вася
```

В реальной жизни массивы объектов – обычное дело, поэтому метод `find` крайне полезен.

Обратите внимание, что в данном примере мы передаём `find` функцию `item => item.id == 1`, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко.

Метод arr.findIndex – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и `-1`, если ничего не найдено.

filter

Метод `find` ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт `true`.

На тот случай, если найденных элементов может быть много, предусмотрен метод arr.filter(fn).

Синтаксис этого метода схож с `find`, но `filter` возвращает массив из всех подходящих элементов:

```
let results = arr.filter(function(item, index, array) {
  // если true - элемент добавляется к результату, и перебор продолжается
  // возвращается пустой массив в случае, если ничего не найдено
});
```

Например:

```
let users = [
  {id: 1, name: "Вася"},
  {id: 2, name: "Петя"},
  {id: 3, name: "Маша"}
];

// возвращает массив, состоящий из двух первых пользователей
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

Преобразование массива

Перейдём к методам преобразования и упорядочения массива.

map

Метод arr.map является одним из наиболее полезных и

часто используемых.

Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции.

Синтаксис:

```
let result = arr.map(function(item, index, array) {
  // возвращается новое значение вместо элемента
});
```

Например, здесь мы преобразуем каждый элемент в его длину:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item =>
item.length);
alert(lengths); // 5,7,6
```

sort(fn)

Вызов arr.sort() сортирует массив *на месте*, меняя в нём порядок элементов.

Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам `arr`.

Например:

```
let arr = [ 1, 2, 15 ];

// метод сортирует содержимое arr
arr.sort();

alert( arr );  //
1, 15, 2
```

Не заметили ничего странного в этом примере?

Порядок стал `1, 15, 2`. Это неправильно! Но почему?

По умолчанию элементы сортируются как строки.

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок, и действительно выходит, что `"2" > "15"`.

Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента `arr.sort()`.

Функция должна для пары значений возвращать:

```
function compare(a, b) {
  if (a > b) return 1; // если первое значение больше второго
  if (a == b) return 0; // если равны
  if (a < b) return -1; // если первое значение меньше второго
}
```

Например, для сортировки чисел:

```
function compareNumeric(a, b) {
  if (a > b) return 1;
  if (a == b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 2, 15 ];



alert(arr);  //
arr.sort(compareNumeric);
1, 2, 15
```

Теперь всё работает как надо.

Давайте возьмём паузу и подумаем, что же происходит. Упомянутый ранее массив `arr` может быть массивом чего угодно, верно? Он может содержать числа, строки, объекты или что-то ещё. У нас есть набор *каких-то элементов*. Чтобы отсортировать его, нам нужна *функция, определяющая порядок*, которая знает, как сравнивать его

элементы. По умолчанию элементы сортируются как строки.

Метод `arr.sort(fn)` реализует общий алгоритм сортировки. Нам не нужно заботиться о том, как он работает внутри (в большинстве случаев это оптимизированная быстрая сортировка). Она проходится по массиву, сравнивает его элементы с помощью предоставленной функции и переупорядочивает их. Всё, что остаётся нам, это предоставить `fn`, которая делает это сравнение.

Кстати, если мы когда-нибудь захотим узнать, какие элементы сравниваются – ничто не мешает нам вывести их на экран:

```
[1, -2, 15, 2, 0, 8].sort(function(a, b) {
  alert( a + " <> " + b );
});
```

В процессе работы алгоритм может сравнивать элемент с другими по нескольку раз, но он старается сделать как можно меньше сравнений.

Функция сравнения может вернуть любое число

На самом деле от функции сравнения требуется любое положительное число, чтобы сказать «больше», и отрицательное число, чтобы сказать «меньше».

Это позволяет писать более короткие функции:

```
let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });
```

```
alert(arr);  //
1, 2, 15
```

Лучше использовать стрелочные функции

Помните стрелочные функции? Можно использовать их здесь для того, чтобы сортировка выглядела более аккуратной:

```
arr.sort( (a, b) => a - b );
```

Будет работать точно так же, как и более длинная версия выше.

reverse

Метод arr.reverse меняет порядок элементов в `arr` на обратный.

Например:

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert( arr ); // 5,4,3,2,1
```

Он также возвращает массив `arr` с изменённым порядком элементов.

split и join

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: `Вася, Петя, Маша`. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Как его получить?

Метод str.split(delim) именно это и делает. Он разбивает строку на массив по заданному разделителю `delim`.

В примере ниже таким разделителем является строка из запятой и пробела.

```
let names = 'Вася, Петя, Маша';

let arr = names.split(', ');

for (let name of arr) {
  alert( `Сообщение получат: ${name}.` ); // Сообщение
получат: Вася (и другие имена)
}
```

У метода `split` есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен. На практике это редко используется:

```
let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2);

alert(arr); // Вася, Петя
```

Разбивка по буквам

Вызов `split(s)` с пустым аргументом `s` разбил бы строку на массив букв:

```
let str = "тест";

alert( str.split('') ); // т,е,с,т
```

Вызов arr.join(glue) делает в точности противоположное `split`. Он создаёт строку из элементов `arr`, вставляя `glue` между ними.

Например:

```
let arr = ['Вася', 'Петя', 'Маша'];

let str = arr.join(';'); // объединить массив в строку через ;

alert( str ); // Вася;Петя;Маша
```

reduce/reduceRight

Если нам нужно перебрать массив – мы можем использовать `forEach`, `for` или `for..of`.

Если нам нужно перебрать массив и вернуть данные для каждого элемента – мы используем `map`.

Методы arr.reduce и arr.reduceRight похожи на методы выше, но они немного сложнее. Они используются для вычисления какого-нибудь единого значения на основе всего массива.

Синтаксис:

```
let value = arr.reduce(function(previousValue, item, index,
array) {
  // ...
}, [initial]);
```

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов.

Аргументы:

- `previousValue` – результат предыдущего вызова этой функции, равен `initial` при первом вызове (если передан `initial`),
- `item` – очередной элемент массива,
- `index` – его индекс,
- `array` – сам массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент.

Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом `reduce`.

Этот метод проще всего понять на примере.

Тут мы получим сумму всех элементов массива всего одной строкой:

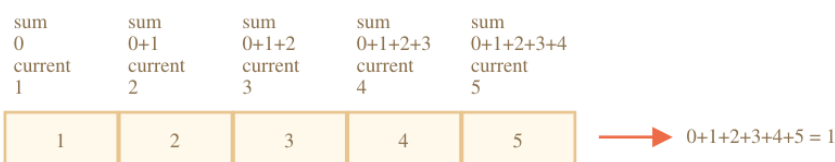
```
let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15
```

Здесь мы использовали наиболее распространённый вариант `reduce`, который использует только 2 аргумента.

Давайте детальнее разберём, как он работает.

1. При первом запуске `sum` равен `initial` (последний аргумент `reduce`), то есть `0`, а `current` – первый элемент массива, равный `1`. Таким образом, результат функции равен `1`.
2. При втором запуске `sum = 1`, и к нему мы добавляем второй элемент массива (`2`).
3. При третьем запуске `sum = 3`, к которому мы добавляем следующий элемент, и так далее…

Поток вычислений получается такой:



В виде таблицы, где каждая строка — вызов функции на очередном элементе массива:

	sum	current	result
первый	0	1	1

вызов			
второй вызов	1	2	3
третий вызов	3	3	6
четвёртый вызов	6	4	10
пятый вызов	10	5	15

Здесь отчётливо видно, как результат предыдущего вызова передаётся в первый аргумент следующего.

Мы также можем опустить начальное значение:

```
let arr = [1, 2, 3, 4, 5];

// убрано начальное значение (нет 0 в конце)
let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15
```

Результат – точно такой же! Это потому, что при отсутствии `initial` в качестве первого значения берётся первый элемент массива, а перебор стартует со второго.

Таблица вычислений будет такая же за вычетом первой строки.

Но такое использование требует крайней осторожности. Если массив пуст, то вызов `reduce` без начального значения выдаст ошибку.

Вот пример:

```
let arr = [];

// Error: Reduce of empty array with no initial value
// если бы существовало начальное значение, reduce вернул бы его для пустого массива.
arr.reduce((sum, current) => sum + current);
```

Поэтому рекомендуется всегда указывать начальное значение.

Метод arr.reduceRight работает аналогично, но проходит по массиву справа налево.

Array.isArray

Массивы не образуют отдельный тип языка. Они основаны на объектах.

Поэтому `typeof` не может отличить простой объект от массива:

```
alert(typeof {}); // object
alert(typeof []); // тоже object
```

…Но массивы используются настолько часто, что для этого придумали специальный метод: Array.isArray(value). Он возвращает `true`, если `value` массив, и `false`, если нет.

```
alert(Array.isArray({})); // false

alert(Array.isArray([])); // true
```

Большинство методов поддерживают «thisArg»

Почти все методы массива, которые вызывают функции – такие как `find`, `filter`, `map`, за исключением метода `sort`, принимают необязательный параметр `thisArg`.

Этот параметр не объяснялся выше, так как очень редко используется, но для наиболее полного понимания темы мы обязаны его рассмотреть.

Вот полный синтаксис этих методов:

```
arr.find(func, thisArg);
arr.filter(func, thisArg);
arr.map(func, thisArg);
// ...
// thisArg - это необязательный последний аргумент
```

Значение параметра `thisArg` становится `this` для `func`.

Например, вот тут мы используем метод объекта `army` как фильтр, и `thisArg` передаёт ему контекст:

```
let army = {
  minAge: 18,
  maxAge: 27,
  canJoin(user) {
    return user.age >= this.minAge && user.age < this.maxAge;
  }
};

let users = [
  {age: 16},
  {age: 20},
  {age: 23},
  {age: 30}
];



alert(soldiers.length); // 2
alert(soldiers[0].age); // 20
alert(soldiers[1].age); // 23
// найти пользователей, для которых
army.canJoin возвращает true
let soldiers = users.filter(army.canJoin,
army);
```

Если бы мы в примере выше использовали просто `users.filter(army.canJoin)`, то вызов `army.canJoin` был бы в режиме отдельной функции, с `this=undefined`. Это тут же привело бы к ошибке.

Вызов `users.filter(army.canJoin, army)` можно заменить на `users.filter(user =>`

army.canJoin(user)), который делает то же самое. Последняя запись используется даже чаще, так как функция-стрелка более наглядна.

Итого

Шпаргалка по методам массива:

Для добавления/удаления элементов:

- `push (...items)` – добавляет элементы в конец,
- `pop()` – извлекает элемент с конца,
- `shift()` – извлекает элемент с начала,
- `unshift(...items)` – добавляет элементы в начало.
- `splice(pos, deleteCount, ...items)` – начиная с индекса `pos`, удаляет `deleteCount` элементов и вставляет `items`.
- `slice(start, end)` – создаёт новый массив, копируя в него элементы с позиции `start` до `end` (не включая `end`).
- `concat(...items)` – возвращает новый массив: копирует все члены текущего массива и добавляет к нему `items`. Если какой-то из `items` является массивом, тогда берутся его элементы.

Для поиска среди элементов:

- `indexOf/lastIndexOf(item, pos)` – ищет `item`, начиная с позиции `pos`, и возвращает его индекс или `-1`, если ничего не найдено.
- `includes(value)` – возвращает `true`, если в массиве имеется элемент `value`, в противном случае `false`.
- `find/filter(func)` – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается

`true`.

`findIndex` похож на `find`, но возвращает индекс вместо значения.

Для перебора элементов:

`forEach(func)` – вызывает `func` для каждого элемента. Ничего не возвращает.

Для преобразования массива:

`map(func)` – создаёт новый массив из результатов вызова `func` для каждого элемента.

`sort(func)` – сортирует массив «на месте», а потом возвращает его.

`reverse()` – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.

`split/join` – преобразует строку в массив и обратно.

`reduce(func, initial)` – вычисляет одно значение на основе всего массива, вызывая `func` для каждого элемента и передавая промежуточный результат между вызовами.

Дополнительно:

`Array.isArray(arr)` проверяет, является ли `arr` массивом.

Обратите внимание, что методы `sort`, `reverse` и `splice` изменяют исходный массив.

Изученных нами методов достаточно в 99% случаев, но существуют и другие.

arr.some(fn)/arr.every(fn) проверяет массив.

Функция `fn` вызывается для каждого элемента массива аналогично `map`. Если какие-либо/все результаты вызовов являются `true`, то метод

возвращает `true`, иначе `false`.

arr.fill(value, start, end) – заполняет массив повторяющимися `value`, начиная с индекса `start` до `end`.

arr.copyWithin(target, start, end) – копирует свои элементы, начиная со `start` и заканчивая `end`, в *собственную* позицию `target` (перезаписывает существующие).

Полный список есть в справочнике MDN.

На первый взгляд может показаться, что существует очень много разных методов, которые довольно сложно запомнить. Но это гораздо проще, чем кажется.

Внимательно изучите шпаргалку, представленную выше, а затем, чтобы попрактиковаться, решите задачи, предложенные в данной главе. Так вы получите необходимый опыт в правильном использовании методов массива.

Всякий раз, когда вам будет необходимо что-то сделать с массивом, а вы не знаете, как это сделать – приходите сюда, смотрите на таблицу и ищите правильный метод. Примеры помогут вам всё сделать правильно, и вскоре вы быстро запомните методы без особых усилий.

Задачи

Переведите текст вида border-left-width в borderLeftWidth

важность: 5

Напишите функцию `camelize(str)`, которая

преобразует строки вида «my-short-string» в «myShortString».

То есть дефисы удаляются, а все слова после них получают заглавную букву.

Примеры:

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

P.S. Подсказка: используйте `split`, чтобы разбить строку на массив символов, потом переделайте всё как нужно и методом `join` соедините обратно.

Открыть песочницу с тестами для задачи.

К решению

Фильтрация по диапазону

важность: 4

Напишите функцию `filterRange(arr, a, b)`, которая принимает массив `arr`, ищет в нём элементы между `a` и `b` и отдаёт массив этих элементов.

Функция должна возвращать новый массив и не изменять исходный.

Например:

```
let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);
```

```
alert( filtered ); // 3,1 (совпадающие значения)

alert( arr ); // 5,3,8,1 (без изменений)
```

Открыть песочницу с тестами для задачи.

К решению

Фильтрация по диапазону "на месте"

важность: 4

Напишите функцию `filterRangeInPlace(arr, a, b)`, которая принимает массив `arr` и удаляет из него все значения кроме тех, которые находятся между `a` и `b`. То есть, проверка имеет вид `a ≤ arr[i] ≤ b`.

Функция должна изменять принимаемый массив и ничего не возвращать.

Например:

```
let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // удалены числа вне диапазона 1..4

alert( arr ); // [3, 1]
```

Открыть песочницу с тестами для задачи.

К решению

Сортировать в порядке по убыванию

важность: 4

```
let arr = [5, 2, 1, -10, 8];

// ... ваш код для сортировки по убыванию

alert( arr ); // 8, 5, 2, 1, -10
```

К решению

Скопировать и отсортировать массив

важность: 5

У нас есть массив строк `arr`. Нужно получить отсортированную копию, но оставить `arr` неизменённым.

Создайте функцию `copySorted(arr)`, которая будет возвращать такую копию.

```
let arr = ["HTML", "JavaScript", "CSS"];

let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript
alert( arr ); // HTML, JavaScript, CSS (без изменений)
```

К решению

Создать расширяемый калькулятор

важность: 5

Создайте функцию конструктор `Calculator`, которая создаёт «расширяемые» объекты калькулятора.

Задание состоит из двух частей.

1. Во-первых, реализуйте метод `calculate(str)`, который принимает строку типа `"1 + 2"` в формате «ЧИСЛО оператор ЧИСЛО» (разделено пробелами) и возвращает результат. Метод должен понимать плюс `+` и минус `-`.

   Пример использования:

   ```
   let calc = new Calculator;

   alert( calc.calculate("3 + 7") ); // 10
   ```

2. Затем добавьте метод `addMethod(name, func)`, который добавляет в калькулятор новые операции. Он принимает оператор `name` и функцию с двумя аргументами `func(a,b)`, которая описывает его.

   Например, давайте добавим умножение `*`, деление `/` и возведение в степень `**`:

   ```
   let powerCalc = new Calculator;
   powerCalc.addMethod("*", (a, b) => a * b);
   powerCalc.addMethod("/", (a, b) => a / b);
   powerCalc.addMethod("**", (a, b) => a ** b);

   let result = powerCalc.calculate("2 ** 3");
   alert( result ); // 8
   ```

   Для этой задачи не нужны скобки или сложные выражения.

   Числа и оператор разделены ровно одним пробелом.

   Не лишним будет добавить обработку ошибок.

Открыть песочницу с тестами для задачи.

К решению

Трансформировать в массив имён

важность: 5

У вас есть массив объектов `user`, и в каждом из них есть `user.name`. Напишите код, который преобразует их в массив имён.

Например:

```
let vasya = { name: "Вася", age: 25 };
let petya = { name: "Петя", age: 30 };
let masha = { name: "Маша", age: 28 };

let users = [ vasya, petya, masha ];

let names = /* ... ваш код */

alert( names ); // Вася, Петя, Маша
```

К решению

Трансформировать в объекты

важность: 5

У вас есть массив объектов `user`, и у каждого из объектов есть `name`, `surname` и `id`.

Напишите код, который создаст ещё один массив объектов с параметрами `id` и `fullName`, где `fullName` – состоит из `name` и `surname`.

Например:

```
let vasya = { name: "Вася", surname: "Пупкин", id: 1 };
let petya = { name: "Петя", surname: "Иванов", id: 2 };
let masha = { name: "Маша", surname: "Петрова", id: 3 };

let users = [ vasya, petya, masha ];



/*
usersMapped = [
  { fullName: "Вася Пупкин", id: 1 },
  { fullName: "Петя Иванов", id: 2 },
  { fullName: "Маша Петрова", id: 3 }
]
*/

alert( usersMapped[0].id ) // 1
alert( usersMapped[0].fullName ) // Вася Пупкин
let usersMapped = /* ... ваш код ... */
```

Итак, на самом деле вам нужно трансформировать один массив объектов в другой. Попробуйте использовать `=>`. Это небольшая уловка.

К решению

Отсортировать пользователей по возрасту

важность: 5

Напишите функцию `sortByAge(users)`, которая принимает массив объектов со свойством `age` и сортирует их по нему.

Например:

```
let vasya = { name: "Вася", age: 25 };
let petya = { name: "Петя", age: 30 };
let masha = { name: "Маша", age: 28 };

let arr = [ vasya, petya, masha ];
```

```
sortByAge(arr);

// теперь: [vasya, masha, petya]
alert(arr[0].name); // Вася
alert(arr[1].name); // Маша
alert(arr[2].name); // Петя
```

К решению

Перемешайте массив

важность: 3

Напишите функцию `shuffle(array)`, которая перемешивает (переупорядочивает случайным образом) элементы массива.

Многократные прогоны через `shuffle` могут привести к разным последовательностям элементов. Например:

```
let arr = [1, 2, 3];

shuffle(arr);
// arr = [3, 2, 1]

shuffle(arr);
// arr = [2, 1, 3]

shuffle(arr);
// arr = [3, 1, 2]
// ...
```

Все последовательности элементов должны иметь одинаковую вероятность. Например, `[1,2,3]` может быть переупорядочено как `[1,2,3]` или `[1,3,2]`, или `[3,1,2]` и т.д., с равной вероятностью каждого случая.

К решению

Получить средний возраст

важность: 4

Напишите функцию `getAverageAge(users)`, которая принимает массив объектов со свойством `age` и возвращает средний возраст.

Формула вычисления среднего арифметического значения: `(age1 + age2 + ... + ageN) / N`.

Например:

```
let vasya = { name: "Вася", age: 25 };
let petya = { name: "Петя", age: 30 };
let masha = { name: "Маша", age: 29 };

let arr = [ vasya, petya, masha ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

К решению

Оставить уникальные элементы массива

важность: 4

Пусть `arr` – массив строк.

Напишите функцию `unique(arr)`, которая возвращает массив, содержащий только уникальные элементы `arr`.

Например:

```
function unique(arr) {
  /* ваш код */
}
```

```
let strings = ["кришна", "кришна", "харе", "харе",
  "харе", "харе", "кришна", "кришна", ":-O"
];

alert( unique(strings) ); // кришна, харе, :-O
```

Открыть песочницу с тестами для задачи.

К решению

Перебираемые объекты

Перебираемые (или *итерируемые*) объекты – это концепция, которая позволяет использовать любой объект в цикле `for..of`.

Конечно же, сами массивы являются перебираемыми объектами. Но есть и много других встроенных перебираемых объектов, например, строки.

Если объект не является массивом, но представляет собой коллекцию каких-то элементов, то удобно использовать цикл `for..of` для их перебора, так что давайте посмотрим, как это сделать.

Symbol.iterator

Мы легко поймём принцип устройства перебираемых объектов, создав один из них.

Например, у нас есть объект. Это не массив, но он выглядит подходящим для `for..of`.

Например, объект `range`, который представляет собой диапазон чисел:

```
let range = {
  from: 1,
  to: 5
};

// Мы хотим, чтобы работал for..of:
// for(let num of range) ... num=1,2,3,4,5
```

Чтобы сделать `range` итерируемым (и позволить `for..of` работать с ним), нам нужно добавить в объект метод с именем `Symbol.iterator` (специальный встроенный `Symbol`, созданный как раз для этого).

1. Когда цикл `for..of` запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть *итератор* – объект с методом `next`.
2. Дальше `for..of` работает *только с этим возвращённым объектом*.
3. Когда `for..of` хочет получить следующее значение, он вызывает метод `next()` этого объекта.
4. Результат вызова `next()` должен иметь вид `{done: Boolean, value: any}`, где `done=true` означает, что итерация закончена, в противном случае `value` содержит очередное значение.

Вот полная реализация `range` с пояснениями:

```
let range = {
  from: 1,
  to: 5
};

// 1. вызов for..of сначала вызывает эту функцию
range[Symbol.iterator] = function() {

  // ...она возвращает объект итератора:
  // 2. Далее, for..of работает только с этим итератором,
```

```
запрашивая у него новые значения
  return {
    current: this.from,
    last: this.to,

    // 3. next() вызывается на каждой итерации цикла for..of
    next() {
      // 4. он должен вернуть значение в виде объекта
{done:.., value :...}
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
};

// теперь работает!
for (let num of range) {
  alert(num); // 1, затем 2, 3, 4, 5
}
```

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

> У самого `range` нет метода `next()`.
>
> Вместо этого другой объект, так называемый «итератор», создаётся вызовом `range[Symbol.iterator]()`, и именно его `next()` генерирует значения.

Таким образом, итератор отделён от самого итерируемого объекта.

Технически мы можем объединить их и использовать сам `range` как итератор, чтобы упростить код.

Например, вот так:

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
```

```
    this.current = this.from;
    return this;
  },

  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};

for (let num of range) {
  alert(num); // 1, затем 2, 3, 4, 5
}
```

Теперь `range[Symbol.iterator]()` возвращает сам объект `range`: у него есть необходимый метод `next()`, и он запоминает текущее состояние итерации в `this.current`. Короче? Да. И иногда такой способ тоже хорош.

Недостаток такого подхода в том, что теперь мы не можем использовать этот объект в двух параллельных циклах `for..of`: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах `for..of`, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

Бесконечные итераторы

Можно сделать бесконечный итератор. Например, `range` будет бесконечным при `range.to = Infinity`. Или мы можем создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

Метод `next` не имеет ограничений, он может

возвращать всё новые и новые значения, это нормально.

Конечно же, цикл `for..of` с таким итерируемым объектом будет бесконечным. Но мы всегда можем прервать его, используя `break`.

Строка – перебираемый объект

Среди встроенных перебираемых объектов наиболее широко используются массивы и строки.

Для строки `for..of` перебирает символы:

```
for (let char of "test") {
  // срабатывает 4 раза: по одному для каждого символа
  alert( char ); // t, затем e, затем s, затем t
}
```

И он работает корректно даже с суррогатными парами!

```
let str = '𝒳😂';
for (let char of str) {
    alert( char ); // 𝒳, а затем 😂
}
```

Явный вызов итератора

Чтобы понять устройство итераторов чуть глубже, давайте посмотрим, как их использовать явно.

Мы будем перебирать строку точно так же, как цикл `for..of`, но вручную, прямыми вызовами. Нижеприведённый код получает строковый итератор и берёт из него значения:

```
let str = "Hello";
```

```
// делает то же самое, что и
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  alert(result.value); // выводит символы один за другим
}
```

Такое редко бывает необходимо, но это даёт нам больше контроля над процессом, чем `for..of`. Например, мы можем разбить процесс итерации на части: перебрать немного элементов, затем остановиться, сделать что-то ещё и потом продолжить.

Итерируемые объекты и псевдомассивы

Есть два официальных термина, которые очень похожи, но в то же время сильно различаются. Поэтому убедитесь, что вы как следует поняли их, чтобы избежать путаницы.

> *Итерируемые объекты* – это объекты, которые реализуют метод `Symbol.iterator`, как было описано выше.
>
> *Псевдомассивы* – это объекты, у которых есть индексы и свойство `length`, то есть, они выглядят как массивы.

При использовании JavaScript в браузере или других окружениях мы можем встретить объекты, которые являются итерируемыми или псевдомассивами, или и тем, и другим.

Например, строки итерируемы (для них работает `for..of`) и являются псевдомассивами (они индексированы и есть `length`).

Но итерируемый объект может не быть псевдомассивом. И

наоборот: псевдомассив может не быть итерируемым.

Например, объект `range` из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и `length`.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

```
let arrayLike = { // есть индексы и свойство length => псевдомассив
  0: "Hello",
  1: "World",
  length: 2
};

// Ошибка (отсутствует Symbol.iterator)
for (let item of arrayLike) {}
```

Что у них общего? И итерируемые объекты, и псевдомассивы – это обычно *не массивы*, у них нет методов `push`, `pop` и т.д. Довольно неудобно, если у нас есть такой объект и мы хотим работать с ним как с массивом. Например, мы хотели бы работать с `range`, используя методы массивов. Как этого достичь?

Array.from

Есть универсальный метод Array.from, который принимает итерируемый объект или псевдомассив и делает из него «настоящий» `Array`. После этого мы уже можем использовать методы массивов.

Например:

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};
```

```
alert(arr.pop()); // World (метод работает)
let arr = Array.from(arrayLike); // (*)
```

`Array.from` в строке `(*)` принимает объект, проверяет, является ли он итерируемым объектом или псевдомассивом, затем создаёт новый массив и копирует туда все элементы.

То же самое происходит с итерируемым объектом:

```
// range взят из примера выше
let arr = Array.from(range);
alert(arr); // 1,2,3,4,5 (преобразование массива через toString работает)
```

Полный синтаксис `Array.from` позволяет указать необязательную «трансформирующую» функцию:

```
Array.from(obj[, mapFn, thisArg])
```

Необязательный второй аргумент может быть функцией, которая будет применена к каждому элементу перед добавлением в массив, а `thisArg` позволяет установить `this` для этой функции.

Например:

```
// range взят из примера выше

// возводим каждое число в квадрат
let arr = Array.from(range, num => num * num);

alert(arr); // 1,4,9,16,25
```

Здесь мы используем `Array.from`, чтобы превратить строку в массив её элементов:

```
let str = '𝒳😂';
```

```
// разбивает строку на массив её элементов
let chars = Array.from(str);

alert(chars[0]); // 𝒳
alert(chars[1]); // 
alert(chars.length); // 2
```

В отличие от `str.split`, этот метод в работе опирается на итерируемость строки, и поэтому, как и `for..of`, он корректно работает с суррогатными парами.

Технически это то же самое, что и:

```
let str = '𝒳';

let chars = []; // Array.from внутри себя выполняет тот же цикл
for (let char of str) {
  chars.push(char);
}

alert(chars);
```

…Но гораздо короче.

Мы можем даже создать `slice`, который поддерживает суррогатные пары:

```
function slice(str, start, end) {
  return Array.from(str).slice(start, end).join('');
}

let str = '𝒳 ';

alert( slice(str, 1, 3) ); //

// а вот встроенный метод не поддерживает суррогатные пары
alert( str.slice(1, 3) ); // мусор (две части различных суррогатных пар)
```

Итого

Объекты, которые можно использовать в цикле `for..of`, называются *итерируемыми*.

Технически итерируемые объекты должны иметь метод `Symbol.iterator`.

Результат вызова `obj[Symbol.iterator]` называется *итератором*. Он управляет процессом итерации.

Итератор должен иметь метод `next()`, который возвращает объект `{done: Boolean, value: any}`, где `done:true` сигнализирует об окончании процесса итерации, в противном случае `value` – следующее значение.

Метод `Symbol.iterator` автоматически вызывается циклом `for..of`, но можно вызвать его и напрямую.

Встроенные итерируемые объекты, такие как строки или массивы, также реализуют метод `Symbol.iterator`.

Строковый итератор знает про суррогатные пары.

Объекты, имеющие индексированные свойства и `length`, называются *псевдомассивами*. Они также могут иметь другие свойства и методы, но у них нет встроенных методов массивов.

Если мы заглянем в спецификацию, мы увидим, что большинство встроенных методов рассчитывают на то, что они будут работать с итерируемыми объектами или псевдомассивами вместо «настоящих» массивов, потому что эти объекты более абстрактны.

`Array.from(obj[, mapFn, thisArg])` создаёт настоящий `Array` из итерируемого объекта или псевдомассива `obj`, и затем мы можем применять к нему методы массивов. Необязательные аргументы `mapFn` и `thisArg` позволяют применять функцию с задаваемым контекстом к каждому элементу.

Map и Set

Сейчас мы знаем о следующих сложных структурах данных:

- Объекты для хранения именованных коллекций.
- Массивы для хранения упорядоченных коллекций.

Но этого не всегда достаточно для решения повседневных задач. Поэтому также существуют `Map` и `Set`.

Map

Map – это коллекция ключ/значение, как и `Object`. Но основное отличие в том, что `Map` позволяет использовать ключи любого типа.

Методы и свойства:

- `new Map()` – создаёт коллекцию.
- `map.set(key, value)` – записывает по ключу `key` значение `value`.
- `map.get(key)` – возвращает значение по ключу или `undefined`, если ключ `key` отсутствует.
- `map.has(key)` – возвращает `true`, если ключ `key` присутствует в коллекции, иначе `false`.
- `map.delete(key)` – удаляет элемент по ключу `key`.
- `map.clear()` – очищает коллекцию от всех элементов.
- `map.size` – возвращает текущее количество элементов.

Например:

```
let map = new Map();
```

```
map.set("1", "str1");    // строка в качестве ключа
map.set(1, "num1");      // цифра как ключ
map.set(true, "bool1");  // булево значение как ключ

// помните, обычный объект Object приводит ключи к строкам?
// Map сохраняет тип ключей, так что в этом случае сохранится
2 разных значения:
alert(map.get(1)); // "num1"
alert(map.get("1")); // "str1"

alert(map.size); // 3
```

Как мы видим, в отличие от объектов, ключи не были приведены к строкам. Можно использовать любые типы данных для ключей.

Map может использовать объекты в качестве ключей.

Например:

```
let john = { name: "John" };

// давайте сохраним количество посещений для каждого
пользователя
let visitsCountMap = new Map();

// объект john - это ключ для значения в объекте Map
visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123
```

Использование объектов в качестве ключей – это одна из известных и часто применяемых возможностей объекта `Map`. При строковых ключах обычный объект `Object` может подойти, но для ключей-объектов – уже нет.

Попробуем заменить `Map` на `Object` в примере выше:

```
let john = { name: "John" };

let visitsCountObj = {}; // попробуем использовать объект

visitsCountObj[john] = 123; // возьмём объект john как ключ

// Вот как это было записано!
```

```
alert( visitsCountObj["[object Object]"] ); // 123
```

Так как `visitsCountObj` – это объект, то все ключи он автоматически преобразует к строке, в итоге получился строковой ключ `"[object Object]"`. Это не то, чего мы хотим.

Как объект `Map` сравнивает ключи

Чтобы сравнивать ключи, объект `Map` использует алгоритм SameValueZero. Это почти такое же сравнение, что и `===`, с той лишь разницей, что `NaN` считается равным `NaN`. Так что `NaN` также может использоваться в качестве ключа.

Этот алгоритм не может быть заменён или модифицирован.

Цепочка вызовов

Каждый вызов `map.set` возвращает объект map, так что мы можем объединить вызовы в цепочку:

```
map.set("1", "str1")
  .set(1, "num1")
  .set(true, "bool1");
```

Перебор Map

Для перебора коллекции `Map` есть 3 метода:

`map.keys()` – возвращает итерируемый объект по

ключам,

`map.values()` – возвращает итерируемый объект по значениям,

`map.entries()` – возвращает итерируемый объект по парам вида `[ключ, значение]`, этот вариант используется по умолчанию в `for..of`.

Например:

```
let recipeMap = new Map([
  ["огурец", 500],
  ["помидор", 350],
  ["лук",    50]
]);

// перебор по ключам (овощи)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // огурец, помидор, лук
}

// перебор по значениям (числа)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}

// перебор по элементам в формате [ключ, значение]
for (let entry of recipeMap) { // то же самое, что и
recipeMap.entries()
  alert(entry); // огурец,500 (и так далее)
}
```

Используется порядок вставки

В отличие от обычных объектов `Object`, в `Map` перебор происходит в том же порядке, в каком происходило добавление элементов.

Кроме этого, `Map` имеет встроенный метод `forEach`, схожий со встроенным методом массивов `Array`:

```
// выполняем функцию для каждой пары (ключ, значение)
recipeMap.forEach((value, key, map) => {
  alert(`${key}: ${value}`); // огурец: 500 и так далее
});
```

Object.entries: Map из Object

При создании `Map` мы можем указать массив (или другой итерируемый объект) с парами ключ-значение для инициализации, как здесь:

```
// массив пар [ключ, значение]
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

Если у нас уже есть обычный объект, и мы хотели бы создать `Map` из него, то поможет встроенный метод Object.entries(obj), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате.

Так что мы можем создать `Map` из обычного объекта следующим образом:

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));
```

Здесь `Object.entries` возвращает массив пар ключ-значение: `[ ["name","John"], ["age", 30] ]`. Это именно то, что нужно для создания `Map`.

Object.fromEntries: Object из Map

Мы только что видели, как создать `Map` из обычного объекта при помощи `Object.entries(obj)`.

Есть метод `Object.fromEntries`, который делает противоположное: получив массив пар вида `[ключ, значение]`, он создаёт из них объект:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

Мы можем использовать `Object.fromEntries`, чтобы получить обычный объект из `Map`.

К примеру, у нас данные в `Map`, но их нужно передать в сторонний код, который ожидает обычный объект.

Вот как это сделать:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);



// готово!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
let obj = Object.fromEntries(map.entries());
// make a plain object (*)
```

Вызов `map.entries()` возвращает массив пар ключ/значение, как раз в нужном формате для `Object.fromEntries`.

Мы могли бы написать строку `(*)` ещё короче:

```
let obj = Object.fromEntries(map); // убрать .entries()
```

Это то же самое, так как `Object.fromEntries` ожидает перебираемый объект в качестве аргумента, не обязательно массив. А перебор `map` как раз возвращает пары ключ/значение, так же, как и `map.entries()`. Так что в итоге у нас будет обычный объект с теми же ключами/значениями, что и в `map`.

Set

Объект `Set` – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы это:

- `new Set(iterable)` – создаёт `Set`, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый `Set`.
- `set.add(value)` – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект `set`.
- `set.delete(value)` – удаляет значение, возвращает `true`, если `value` было в множестве на момент вызова, иначе `false`.
- `set.has(value)` – возвращает `true`, если значение присутствует в множестве, иначе `false`.
- `set.clear()` – удаляет все имеющиеся значения.

`set.size` – возвращает количество элементов в множестве.

Основная «изюминка» – это то, что при повторных вызовах `set.add()` с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

Например, мы ожидаем посетителей, и нам необходимо составить их список. Но повторные визиты не должны приводить к дубликатам. Каждый посетитель должен появиться в списке только один раз.

Множество `Set` – как раз то, что нужно для этого:

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set хранит только 3 уникальных значения
alert(set.size); // 3

for (let user of set) {
  alert(user.name); // John (потом Pete и Mary)
}
```

Альтернативой множеству `Set` может выступать массив для хранения гостей и дополнительный код для проверки уже имеющегося элемента с помощью arr.find. Но в этом случае будет хуже производительность, потому что `arr.find` проходит весь массив для проверки наличия элемента. Множество `Set` лучше оптимизировано для добавлений, оно автоматически проверяет на уникальность.

Перебор объекта Set

Мы можем перебрать содержимое объекта set как с помощью метода `for..of`, так и используя `forEach`:

```
let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

// то же самое с forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Заметим забавную вещь. Функция в `forEach` у `Set` имеет 3 аргумента: значение `value`, потом *снова то же самое значение* `valueAgain`, и только потом целевой объект. Это действительно так, значение появляется в списке аргументов дважды.

Это сделано для совместимости с объектом `Map`, в котором колбэк `forEach` имеет 3 аргумента. Выглядит немного странно, но в некоторых случаях может помочь легко заменить `Map` на `Set` и наоборот.

`Set` имеет те же встроенные методы, что и `Map`:

- `set.values()` – возвращает перебираемый объект для значений,
- `set.keys()` – то же самое, что и `set.values()`, присутствует для обратной совместимости с `Map`,
- `set.entries()` – возвращает перебираемый объект для пар вида `[значение, значение]`, присутствует для обратной совместимости с `Map`.

Итого

`Map` – коллекция пар ключ-значение.

Методы и свойства:

- `new Map([iterable])` – создаёт коллекцию, можно указать перебираемый объект (обычно массив) из пар `[ключ,значение]` для инициализации.
- `map.set(key, value)` – записывает по ключу `key` значение `value`.
- `map.get(key)` – возвращает значение по ключу или `undefined`, если ключ `key` отсутствует.
- `map.has(key)` – возвращает `true`, если ключ `key` присутствует в коллекции, иначе `false`.
- `map.delete(key)` – удаляет элемент по ключу `key`.
- `map.clear()` – очищает коллекцию от всех элементов.
- `map.size` – возвращает текущее количество элементов.

Отличия от обычного объекта `Object`:

- Что угодно может быть ключом, в том числе и объекты.
- Есть дополнительные методы, свойство `size`.

`Set` – коллекция уникальных значений, так называемое «множество».

Методы и свойства:

- `new Set([iterable])` – создаёт `Set`, можно указать перебираемый объект со значениями для инициализации.
- `set.add(value)` – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект `set`.
- `set.delete(value)` – удаляет значение, возвращает `true` если `value` было в множестве на момент вызова, иначе `false`.
- `set.has(value)` – возвращает `true`, если значение присутствует в множестве, иначе `false`.

`set.clear()` – удаляет все имеющиеся значения.

`set.size` – возвращает количество элементов в множестве.

Перебор `Map` и `Set` всегда осуществляется в порядке добавления элементов, так что нельзя сказать, что это – неупорядоченные коллекции, но поменять порядок элементов или получить элемент напрямую по его номеру нельзя.

Задачи

Фильтрация уникальных элементов массива

важность: 5

Допустим, у нас есть массив `arr`.

Создайте функцию `unique(arr)`, которая вернёт массив уникальных, не повторяющихся значений массива `arr`.

Например:

```
function unique(arr) {
  /* ваш код */
}

let values = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(values) ); // Hare,Krishna,:-O
```

P.S. Здесь мы используем строки, но значения могут быть любого типа.

P.P.S. Используйте `Set` для хранения уникальных значений.

Открыть песочницу с тестами для задачи.

К решению

Отфильтруйте анаграммы

важность: 4

Анаграммы – это слова, у которых те же буквы в том же количестве, но они располагаются в другом порядке.

Например:

```
nap - pan
ear - are - era
cheaters - hectares - teachers
```

Напишите функцию `aclean(arr)`, которая возвращает массив слов, очищенный от анаграмм.

Например:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era",
"hectares"];

alert( aclean(arr) ); // "nap,teachers,ear" or
"PAN,cheaters,era"
```

Из каждой группы анаграмм должно остаться только одно слово, не важно какое.

Открыть песочницу с тестами для задачи.

К решению

Перебираемые ключи

важность: 5

Мы хотели бы получить массив ключей `map.keys()` в переменную и далее работать с ними, например, применить метод `.push`.

Но это не выходит:

```
let map = new Map();

map.set("name", "John");

let keys = map.keys();

// Error: keys.push is not a function
// Ошибка: keys.push -- это не функция
keys.push("more");
```

Почему? Что нужно поправить в коде, чтобы вызов `keys.push` сработал?

К решению

WeakMap и WeakSet

Как мы знаем из главы Сборка мусора, движок JavaScript хранит значения в памяти до тех пор, пока они достижимы (то есть, эти значения могут быть использованы).

Например:

```
let john = { name: "John" };

// объект доступен, переменная john -- это ссылка на него

// перепишем ссылку
john = null;

// объект будет удалён из памяти
```

Обычно свойства объекта, элементы массива или другой структуры данных считаются достижимыми и сохраняются в памяти до тех пор, пока эта структура данных содержится в памяти.

Например, если мы поместим объект в массив, то до тех пор, пока массив существует, объект также будет существовать в памяти, несмотря на то, что других ссылок на него нет.

Например:

```
let john = { name: "John" };

let array = [ john ];

john = null; // перезаписываем ссылку на объект

// объект john хранится в массиве, поэтому он
не будет удалён сборщиком мусора
// мы можем взять его значение как array[0]
```

Аналогично, если мы используем объект как ключ в `Map`, то до тех пор, пока существует `Map`, также будет существовать и этот объект. Он занимает место в памяти и не может быть удалён сборщиком мусора.

Например:

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");
```

```
john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`,
// он доступен через map.keys()
```

`WeakMap` – принципиально другая структура в этом аспекте. Она не предотвращает удаление объектов сборщиком мусора, когда эти объекты выступают в качестве ключей.

Давайте посмотрим, что это означает, на примерах.

WeakMap

Первое его отличие от `Map` в том, что ключи в `WeakMap` должны быть объектами, а не примитивными значениями:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // работает (объект в качестве ключа)

// нельзя использовать строку в качестве ключа
weakMap.set("test", "Whoops"); // Ошибка,
потому что "test" не объект
```

Теперь, если мы используем объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удалён из памяти (и из объекта `WeakMap`) автоматически.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john удалён из памяти!
```

Сравните это поведение с поведением обычного `Map`, пример которого был приведён ранее. Теперь `john` существует только как ключ в `WeakMap` и может быть удалён оттуда автоматически.

`WeakMap` не поддерживает перебор и методы `keys()`, `values()`, `entries()`, так что нет способа взять все ключи или значения из неё.

В `WeakMap` присутствуют только следующие методы:

```
weakMap.get(key)
weakMap.set(key, value)
weakMap.delete(key)
weakMap.has(key)
```

К чему такие ограничения? Из-за особенностей технической реализации. Если объект станет недостижим (как объект `john` в примере выше), то он будет автоматически удалён сборщиком мусора. Но нет информации, *в какой момент произойдёт эта очистка*.

Решение о том, когда делать сборку мусора, принимает движок JavaScript. Он может посчитать необходимым как удалить объект прямо сейчас, так и отложить эту операцию, чтобы удалить большее количество объектов за раз позже. Так что технически количество элементов в коллекции `WeakMap` неизвестно. Движок может произвести очистку сразу или потом, или сделать это частично. По этой причине методы для доступа ко всем сразу ключам/значениям недоступны.

Но для чего же нам нужна такая структура данных?

Пример: дополнительные данные

В основном, `WeakMap` используется в качестве

дополнительного хранилища данных.

Если мы работаем с объектом, который «принадлежит» другому коду, может быть даже сторонней библиотеке, и хотим сохранить у себя какие-то данные для него, которые должны существовать лишь пока существует этот объект, то `WeakMap` – как раз то, что нужно.

Мы кладём эти данные в `WeakMap`, используя объект как ключ, и когда сборщик мусора удалит объекты из памяти, ассоциированные с ними данные тоже автоматически исчезнут.

```
weakMap.set(john, "секретные документы");
// если john умрёт, "секретные документы" будут автоматически уничтожены
```

Давайте рассмотрим один пример.

Предположим, у нас есть код, который ведёт учёт посещений для пользователей. Информация хранится в коллекции `Map`: объект, представляющий пользователя, является ключом, а количество визитов – значением. Когда пользователь нас покидает (его объект удаляется сборщиком мусора), то больше нет смысла хранить соответствующий счётчик посещений.

Вот пример реализации счётчика посещений с использованием `Map`:

```
// visitsCount.js
let visitsCountMap = new Map(); // map: пользователь => число визитов

// увеличиваем счётчик
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

А вот другая часть кода, возможно, в другом файле,

которая использует `countUser`:

```
// 📄 main.js
let john = { name: "John" };

countUser(john); //ведём подсчёт посещений

// пользователь покинул нас
john = null;
```

Теперь объект `john` должен быть удалён сборщиком мусора, но он продолжает оставаться в памяти, так как является ключом в `visitsCountMap`.

Нам нужно очищать `visitsCountMap` при удалении объекта пользователя, иначе коллекция будет бесконечно расти. Подобная очистка может быть неудобна в реализации при сложной архитектуре приложения.

Проблемы можно избежать, если использовать `WeakMap`:

```
// 📄 visitsCount.js
let visitsCountMap = new WeakMap(); // map: пользователь =>
число визитов

// увеличиваем счётчик
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

Теперь нет необходимости вручную очищать `visitsCountMap`. После того, как объект `john` стал недостижим другими способами, кроме как через `WeakMap`, он удаляется из памяти вместе с информацией по такому ключу из `WeakMap`.

Применение для кеширования

Другая частая сфера применения – это кеширование,

когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его.

Для хранения результатов мы можем использовать `Map`, вот так:

```
// 📁 cache.js
let cache = new Map();

// вычисляем и запоминаем результат
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* тут какие-то вычисления результата для
объекта */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}



// 📁 main.js
let obj = {/* допустим, у нас есть какой-то объект */};

let result1 = process(obj); // вычислен результат

// ...позже, из другого места в коде...
let result2 = process(obj); // ранее вычисленный результат
взят из кеша

// ...позже, когда объект больше не нужен:
obj = null;

alert(cache.size); // 1 (Упс! Объект всё ещё в кеше, занимает
память!)
// Теперь используем process() в другом файле:
```

Многократные вызовы `process(obj)` с тем же самым объектом в качестве аргумента ведут к тому, что результат вычисляется только в первый раз, а затем последующие вызовы берут его из кеша. Недостатком является то, что необходимо вручную очищать `cache` от ставших

ненужными объектов.

Но если мы будем использовать `WeakMap` вместо `Map`, то эта проблема исчезнет: закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

```
// 📄 cache.js


// вычисляем и запоминаем результат
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* вычисляем результат для объекта */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// 📄 main.js
let obj = {/* какой-то объект */};

let result1 = process(obj);
let result2 = process(obj);

// ...позже, когда объект больше не нужен:
obj = null;

// Нет возможности получить cache.size, так как это WeakMap,
// но он равен 0 или скоро будет равен 0
// Когда сборщик мусора удаляет obj, связанные с ним данные из
кеша тоже удаляются
let cache = new WeakMap();
```

WeakSet

Коллекция `WeakSet` ведёт себя похоже:

- Она аналогична `Set`, но мы можем добавлять в `WeakSet` только объекты (не примитивные значения).
- Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.
- Как и `Set`, она поддерживает `add`, `has` и `delete`, но

не `size`, `keys()` и не является перебираемой.

Будучи «слабой» версией оригинальной структуры данных, она тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве `WeakSet` может что-то сказать нам об объекте.

Например, мы можем добавлять пользователей в `WeakSet` для учёта тех, кто посещал наш сайт:

```
let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // John заходил к нам
visitedSet.add(pete); // потом Pete
visitedSet.add(john); // John снова

// visitedSet сейчас содержит двух пользователей

// проверим, заходил ли John?
alert(visitedSet.has(john)); // true

// проверим, заходила ли Mary?
alert(visitedSet.has(mary)); // false

john = null;

// структура данных visitedSet будет очищена автоматически
```

Наиболее значительным ограничением `WeakMap` и `WeakSet` является то, что их нельзя перебрать или взять всё содержимое. Это может доставлять неудобства, но не мешает `WeakMap/WeakSet` выполнять их главную задачу – быть дополнительным хранилищем данных для объектов, управляемых из каких-то других мест в коде.

Итого

`WeakMap` – это `Map`-подобная коллекция, позволяющая использовать в качестве ключей только объекты, и автоматически удаляющая их вместе с соответствующими значениями, как только они становятся недостижимыми иными путями.

`WeakSet` – это `Set`-подобная коллекция, которая хранит только объекты и удаляет их, как только они становятся недостижимыми иными путями.

Обе этих структуры данных не поддерживают методы и свойства, работающие со всем содержимым сразу или возвращающие информацию о размере коллекции. Возможны только операции на отдельном элементе коллекции.

`WeakMap` и `WeakSet` используются как вспомогательные структуры данных в дополнение к «основному» месту хранения объекта. Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в `WeakMap` или в `WeakSet`, то он будет удалён автоматически.

Задачи

Хранение отметок "не прочитано"

важность: 5

Есть массив сообщений:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];
```

У вас есть к ним доступ, но управление этим массивом

происходит где-то ещё. Добавляются новые сообщения и удаляются старые, и вы не знаете в какой момент это может произойти.

Имея такую вводную информацию, решите, какую структуру данных вы могли бы использовать для ответа на вопрос «было ли сообщение прочитано?». Структура должна быть подходящей, чтобы можно было однозначно сказать, было ли прочитано это сообщение для каждого объекта сообщения.

P.S. Когда сообщение удаляется из массива `messages`, оно должно также исчезать из структуры данных.

P.P.S. Нам не следует модифицировать сами объекты сообщений, добавлять туда свойства. Если сообщения принадлежат какому-то другому коду, то это может привести к плохим последствиям.

К решению

Хранение времени прочтения

важность: 5

Есть массив сообщений такой же, как и в предыдущем задании.

```
let messages = [
  { text: "Hello", from: "John" },
  { text: "How goes?", from: "John" },
  { text: "See you soon", from: "Alice" }
];
```

Теперь вопрос стоит так: какую структуру данных вы бы предложили использовать для хранения информации о

том, когда сообщение было прочитано?

В предыдущем задании нам нужно было сохранить только факт прочтения «да или нет». Теперь же нам нужно сохранить дату, и она должна исчезнуть из памяти при удалении «сборщиком мусора» сообщения.

P.S. Даты в JavaScript можно хранить как объекты встроенного класса `Date`, которые мы разберём позднее.

К решению

Object.keys, values, entries

Давайте отойдём от отдельных структур данных и поговорим об их переборе вообще.

В предыдущей главе мы видели методы `map.keys()`, `map.values()`, `map.entries()`.

Это универсальные методы, и существует общее соглашение использовать их для структур данных. Если бы мы делали собственную структуру данных, нам также следовало бы их реализовать.

Методы поддерживаются для структур:

- `Map`
- `Set`
- `Array`

Простые объекты также можно перебирать похожими

методами, но синтаксис немного отличается.

Object.keys, values, entries

Для простых объектов доступны следующие методы:

- Object.keys(obj) – возвращает массив ключей.
- Object.values(obj) – возвращает массив значений.
- Object.entries(obj) – возвращает массив пар `[ключ, значение]`.

Обратите внимание на различия (по сравнению с `map`, например):

	Map	Object
Синтаксис вызова	`map.keys()`	`Object.keys(obj)`, не `obj.keys()`
Возвращает	перебираемый объект	«реальный» массив

Первое отличие в том, что мы должны вызвать `Object.keys(obj)`, а не `obj.keys()`.

Почему так? Основная причина – гибкость. Помните, что объекты являются основой всех сложных структур в JavaScript. У нас может быть объект `data`, который реализует свой собственный метод `data.values()`. И мы всё ещё можем применять к нему стандартный метод `Object.values(data)`.

Второе отличие в том, что методы вида `Object.*` возвращают «реальные» массивы, а не просто итерируемые объекты. Это в основном по историческим причинам.

Например:

```
let user = {
  name: "John",
  age: 30
};
```

```
Object.keys(user) = ["name", "age"]
Object.values(user) = ["John", 30]
Object.entries(user) = [ ["name","John"], ["age",30] ]
```

Вот пример использования `Object.values` для перебора значений свойств в цикле:

```
let user = {
  name: "John",
  age: 30
};

// перебор значений
for (let value of Object.values(user)) {
  alert(value); // John, затем 30
}
```

Object.keys/values/entries игнорируют символьные свойства

Так же, как и цикл `for..in`, эти методы игнорируют свойства, использующие `Symbol(...)` в качестве ключей.

Обычно это удобно. Но если требуется учитывать и символьные ключи, то для этого существует отдельный метод Object.getOwnPropertySymbols, возвращающий массив только символьных ключей. Также, существует метод Reflect.ownKeys(obj), который возвращает *все*

ключи.

Трансформации объекта

У объектов нет множества методов, которые есть в массивах, например `map`, `filter` и других.

Если мы хотели бы их применить, то можно использовать `Object.entries` с последующим вызовом `Object.fromEntries`:

1. Вызов `Object.entries(obj)` возвращает массив пар ключ/значение для `obj`.
2. На нём вызываем методы массива, например, `map`.
3. Используем `Object.fromEntries(array)` на результате, чтобы преобразовать его обратно в объект.

Например, у нас есть объект с ценами, и мы хотели бы их удвоить:

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
};
```

```
alert(doublePrices.meat); // 8
let doublePrices = Object.fromEntries(
  // преобразовать в массив, затем map, затем
fromEntries обратно объект
  Object.entries(prices).map(([key, value]) =>
[key, value * 2])
);
```

Это может выглядеть сложным на первый взгляд, но становится лёгким для понимания после нескольких раз

использования.

Можно делать и более сложные «однострочные» преобразования таким путём. Важно только сохранять баланс, чтобы код при этом был достаточно простым для понимания.

Задачи

Сумма свойств объекта

важность: 5

Есть объект `salaries` с произвольным количеством свойств, содержащих заработные платы.

Напишите функцию `sumSalaries(salaries)`, которая возвращает сумму всех зарплат с помощью метода `Object.values` и цикла `for..of`.

Если объект `salaries` пуст, то результат должен быть `0`.

Например:

```
let salaries = {
  "John": 100,
  "Pete": 300,
  "Mary": 250
};

alert( sumSalaries(salaries) ); // 650
```

Открыть песочницу с тестами для задачи.

К решению

Подсчёт количества свойств объекта

важность: 5

Напишите функцию `count(obj)`, которая возвращает

количество свойств объекта:

```
let user = {
  name: 'John',
  age: 30
};

alert( count(user) ); // 2
```

Постарайтесь сделать код как можно короче.

P.S. Игнорируйте символьные свойства, подсчитывайте только «обычные».

Открыть песочницу с тестами для задачи.

К решению

Деструктурирующее присваивание

В JavaScript есть две чаще всего используемые структуры данных – это `Object` и `Array`.

Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в кучу переменных, так как иногда они более

удобны. Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию и так далее.

Деструктуризация массива

Пример деструктуризации массива:

```
// у нас есть массив с именем и фамилией
let arr = ["Ilya", "Kantor"]


alert(firstName); // Ilya
alert(surname);  // Kantor
// деструктурирующее присваивание
// записывает firstName=arr[0], surname=arr[1]
let [firstName, surname] = arr;
```

Теперь мы можем использовать переменные вместо элементов массива.

Отлично смотрится в сочетании со `split` или другими методами, возвращающими массив:

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

«Деструктуризация» не означает «разрушение».

«Деструктурирующее присваивание» не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.

Это просто короткий вариант записи:

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

Пропускайте элементы, используя запятые

Ненужные элементы массива также могут быть отброшены через запятую:

```
alert( title ); // Consul
// второй элемент не нужен
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
```

В примере выше второй элемент массива пропускается, а третий присваивается переменной `title`, оставшиеся элементы массива также пропускаются (так как для них нет переменных).

Работает с любым перебираемым объектом с правой стороны

…На самом деле мы можем использовать любой перебираемый объект, не только массивы:

```
let [a, b, c] = "abc";
let [one, two, three] = new Set([1, 2, 3]);
```

Присваивайте чему угодно с левой стороны

Мы можем использовать что угодно «присваивающее» с левой стороны.

Например, можно присвоить свойству объекта:

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split('
');

alert(user.name); // Ilya
```

Цикл с .entries()

В предыдущей главе мы видели метод Object.entries(obj).

Мы можем использовать его с деструктуризацией для цикличного перебора ключей и значений объекта:

```
let user = {
  name: "John",
  age: 30
};

// цикл по ключам и значениям

  alert(`${key}:${value}`); // name:John, затем
age:30
}
for (let [key, value] of
Object.entries(user)) {
```

…то же самое для map:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");


  alert(`${key}:${value}`); // name:John, затем
age:30
}
for (let [key, value] of user) {
```

Остаточные параметры «...»

Если мы хотим не просто получить первые значения, но и собрать все остальные, то мы можем добавить ещё один параметр, который получает остальные значения, используя оператор «остаточные параметры» – троеточие (`"..."`):

```
let [name1, name2, ] = ["Julius", "Caesar", ];

alert(name1); // Julius
alert(name2); // Caesar
```

```
...
rest
"Consul"
, "of the Roman Republic"
// Обратите внимание, что `rest` является
массивом.
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

Переменная `rest` является массивом из оставшихся элементов. Вместо `rest` можно использовать любое другое название переменной, просто убедитесь, что перед переменной есть три точки и она стоит на последнем месте в деструктурирующем присваивании.

Значения по умолчанию

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения считаются

неопределёнными:

```
alert(firstName); // undefined
alert(surname); // undefined
let [firstName, surname] = [];
```

Если нам необходимо указать значения по умолчанию, то мы можем использовать =:

```
alert(name);    // Julius (из массива)
alert(surname); // Anonymous (значение по умолчанию)
// значения по умолчанию
let [name = "Guest", surname = "Anonymous"] =
["Julius"];
```

Значения по умолчанию могут быть гораздо более сложными выражениями или даже функциями. Они выполняются, только если значения отсутствуют.

Например, здесь мы используем функцию `prompt` для указания двух значений по умолчанию. Но она будет запущена только для отсутствующего значения:

```
// prompt запустится только для surname
let [name = prompt('name?'), surname = prompt('surname?')] =
["Julius"];

alert(name);    // Julius (из массива)
alert(surname); // результат prompt
```

Деструктуризация объекта

Деструктурирующее присваивание также работает с объектами.

Синтаксис:

```
let {var1, var2} = {var1:…, var2:…}
```

У нас есть существующий объект с правой стороны, который мы хотим разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в `{...}`.

Например:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};
```

```
alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
let {title, width, height} = options;
```

Свойства `options.title`, `options.width` и `options.height` присваиваются соответствующим переменным. Порядок не имеет значения. Вот так – тоже работает:

```
// изменён порядок в let {...}
let {height, width, title} = { title: "Menu", height: 200,
width: 100 }
```

Шаблон с левой стороны может быть более сложным и определять соответствие между свойствами и переменными.

Если мы хотим присвоить свойство объекта переменной с другим названием, например, свойство `options.width`

присвоить переменной `w`, то мы можем использовать двоеточие:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};



// width -> w
// height -> h
// title -> title

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;
```

Двоеточие показывает «что : куда идёт». В примере выше свойство `width` сохраняется в переменную `w`, свойство `height` сохраняется в `h`, а `title` присваивается одноимённой переменной.

Для потенциально отсутствующих свойств мы можем установить значения по умолчанию, используя `"="`, как здесь:

```
let options = {
  title: "Menu"
};



alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
let {width = 100, height = 200, title} =
options;
```

Как и в случае с массивами, значениями по умолчанию

могут быть любые выражения или даже функции. Они выполнятся, если значения отсутствуют.

В коде ниже `prompt` запросит `width`, но не `title`:

```
let options = {
  title: "Menu"
};



alert(title);  // Menu
alert(width);  // (результат prompt)
let {width = prompt("width?"), title =
prompt("title?")} = options;
```

Мы также можем совмещать `:` и `=`:

```
let options = {
  title: "Menu"
};



alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
let {width: w = 100, height: h = 200, title} =
options;
```

Если у нас есть большой объект с множеством свойств, можно взять только то, что нужно:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// взять только title, игнорировать остальное
let { title } = options;

alert(title); // Menu
```

Остаток объекта «...»

Что если в объекте больше свойств, чем у нас переменных? Можем ли мы взять необходимые нам, а остальные присвоить куда-нибудь?

Можно использовать троеточие, как и для массивов. В некоторых старых браузерах (IE) это не поддерживается, используйте Babel для полифила.

Выглядит так:

```
let options = {
  title: "Menu",
  height: 200,
  width: 100
};
```

```
// сейчас title="Menu", rest={height: 200, width: 100}
alert(rest.height);  // 200
alert(rest.width);   // 100
// title = свойство с именем title
// rest = объект с остальными свойствами
let {title, ...rest} = options;
```

Обратите внимание на `let`

В примерах выше переменные были объявлены в присваивании: `let {…} = {…}`. Конечно, мы могли бы использовать существующие переменные и не указывать `let`, но тут есть подвох.

Вот так не будет работать:

```
let title, width, height;

// ошибка будет в этой строке
{title, width, height} = {title: "Menu", width: 200,
```

```
height: 100};
```

Проблема в том, что JavaScript обрабатывает `{...}` в основном потоке кода (не внутри другого выражения) как блок кода. Такие блоки кода могут быть использованы для группировки операторов, например:

```
{
  // блок кода
  let message = "Hello";
  // ...
  alert( message );
}
```

Так что здесь JavaScript считает, что видит блок кода, отсюда и ошибка. На самом-то деле у нас деструктуризация.

Чтобы показать JavaScript, что это не блок кода, мы можем заключить выражение в скобки `(...)`:

```
let title, width, height;

// сейчас всё работает
{title, width, height} = {title: "Menu", width: 200,
height: 100};

alert( title ); // Menu
(
)
```

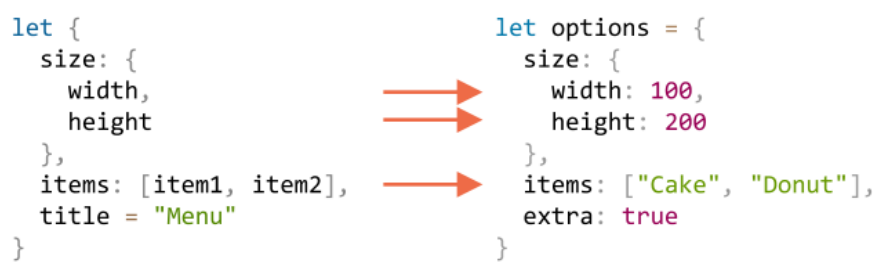
Вложенная деструктуризация

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства.

В приведённом ниже коде `options` хранит другой объект в свойстве `size` и массив в свойстве `items`. Шаблон в левой части присваивания имеет такую же структуру, чтобы извлечь данные из них:

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

// деструктуризация разбита на несколько строк для ясности
let {
  size: { // положим size сюда
    width,
    height
  },
  items: [item1, item2], // добавим элементы к items
  title = "Menu" // отсутствует в объекте (используется
значение по умолчанию)
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

Весь объект `options`, кроме свойства `extra`, которое в левой части отсутствует, присваивается в соответствующие переменные:



В итоге у нас есть `width`, `height`, `item1`, `item2` и

`title` со значением по умолчанию.

Заметим, что переменные для `size` и `items` отсутствуют, так как мы взяли сразу их содержимое.

Умные параметры функций

Есть ситуации, когда функция имеет много параметров, большинство из которых не обязательны. Это особенно верно для пользовательских интерфейсов. Представьте себе функцию, которая создаёт меню. Она может иметь ширину, высоту, заголовок, список элементов и так далее.

Вот так – плохой способ писать подобные функции:

```
function showMenu(title = "Untitled", width = 200, height =
100, items = []) {
  // ...
}
```

В реальной жизни проблема заключается в том, как запомнить порядок всех аргументов. Обычно IDE пытаются помочь нам, особенно если код хорошо документирован, но всё же… Другая проблема заключается в том, как вызвать функцию, когда большинство параметров передавать не надо, и значения по умолчанию вполне подходят.

Разве что вот так?

```
// undefined там, где подходят значения по умолчанию
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

Это выглядит ужасно. И становится нечитаемым, когда мы имеем дело с большим количеством параметров.

На помощь приходит деструктуризация!

Мы можем передать параметры как объект, и функция немедленно деструктурирует его в переменные:

```
// мы передаём объект в функцию
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// ...и она немедленно извлекает свойства в переменные
function showMenu() {
  // title, items – взято из options,
  // width, height – используются значения по умолчанию
  alert( `${title} ${width} ${height}` ); // My Menu 200 100
  alert( items ); // Item1, Item2
}

showMenu(options);
{
title = "Untitled", width = 200, height = 100,
items = []}
```

Мы также можем использовать более сложное деструктурирование с вложенными объектами и двоеточием:

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};


  alert( `${title} ${w} ${h}` ); // My Menu 100 200
  alert( item1 ); // Item1
  alert( item2 ); // Item2
}

showMenu(options);
function showMenu({
  title = "Untitled",
  width: w = 100,  // width присваиваем в w
  height: h = 200, // height присваиваем в h
  items: [item1, item2] // первый элемент
items присваивается в item1, второй в item2
}) {
```

Полный синтаксис – такой же, как для деструктурирующего присваивания:

```
function({
  incomingProperty: varName = defaultValue
  ...
})
```

Тогда для объекта с параметрами будет создана переменная `varName` для свойства с именем `incomingProperty` по умолчанию равная `defaultValue`.

Пожалуйста, обратите внимание, что такое деструктурирование подразумевает, что в `showMenu()` будет обязательно передан аргумент. Если нам нужны все значения по умолчанию, то нам следует передать пустой объект:

```
showMenu({}); // ок, все значения - по умолчанию

showMenu(); // так была бы ошибка
```

Мы можем исправить это, сделав `{}` значением по умолчанию для всего объекта параметров:

```
function showMenu({ title = "Menu", width = 100, height = 200
}) {
  alert( `${title} ${width} ${height}` );
}

showMenu(); // Menu 100 200
 = {}
```

В приведённом выше коде весь объект аргументов по умолчанию равен `{}`, поэтому всегда есть что-то, что можно деструктурировать.

Итого

Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.

Полный синтаксис для объекта:

```
let {prop : varName = default, ...rest} = object
```

Свойства, которые не были упомянуты, копируются в объект `rest`.

Полный синтаксис для массива:

```
let [item1 = default, item2, ...rest] = array
```

Первый элемент отправляется в `item1`; второй отправляется в `item2`, все остальные элементы попадают в массив `rest`.

Можно извлекать данные из вложенных объектов и массивов, для этого левая сторона должна иметь ту же структуру, что и правая.

Задачи

Деструктурирующее присваивание

важность: 5

У нас есть объект:

```
let user = {
  name: "John",
  years: 30
};
```

Напишите деструктурирующее присваивание, которое:

свойство `name` присвоит в переменную `name`.

свойство `years` присвоит в переменную `age`.

свойство `isAdmin` присвоит в переменную `isAdmin` (false, если нет такого свойства)

Пример переменных после вашего присваивания:

```
let user = { name: "John", years: 30 };

// ваш код должен быть с левой стороны:
// ... = user

alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

К решению

Максимальная зарплата

важность: 5

У нас есть объект `salaries` с зарплатами:

```
let salaries = {
  "John": 100,
  "Pete": 300,
  "Mary": 250
};
```

Создайте функцию `topSalary(salaries)`, которая возвращает имя самого высокооплачиваемого сотрудника.

Если объект `salaries` пустой, то нужно вернуть `null`.

Если несколько высокооплачиваемых сотрудников, можно вернуть любого из них.

P.S. Используйте `Object.entries` и деструктурирование, чтобы перебрать пары ключ/значение.

Открыть песочницу с тестами для задачи.

К решению

Дата и время

Встречайте новый встроенный объект: Date. Он содержит дату и время, а также предоставляет методы управления ими.

Например, его можно использовать для хранения времени создания/изменения, для измерения времени или просто для вывода текущей даты.

Создание

Для создания нового объекта `Date` нужно вызвать конструктор `new Date()` с одним из следующих аргументов:

`new Date()`

Без аргументов – создать объект `Date` с текущими датой и временем:

```
let now = new Date();
alert( now ); // показывает текущие дату и время
```

new Date(milliseconds)

Создать объект `Date` с временем, равным количеству миллисекунд (тысячная доля секунды), прошедших с 1 января 1970 года UTC+0.

```
// 0 соответствует 01.01.1970 UTC+0
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );

// теперь добавим 24 часа и получим 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

Целое число, представляющее собой количество миллисекунд, прошедших с начала 1970 года, называется *таймстамп* (англ. timestamp).

Это – легковесное численное представление даты. Из таймстампа всегда можно получить дату с помощью `new Date(timestamp)` и преобразовать существующий объект `Date` в таймстамп, используя метод `date.getTime()` (см. ниже).

Датам до 1 января 1970 будут соответствовать отрицательные таймстампы, например:

```
// 31 декабря 1969 года
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

new Date(datestring)

Если аргумент всего один, и это строка, то из неё «прочитывается» дата. Алгоритм разбора – такой же, как в `Date.parse`, который мы рассмотрим позже.

```
let date = new Date("2017-01-26");
alert(date);
// Время не указано, поэтому оно ставится в полночь по
Гринвичу и
// меняется в соответствии с часовым поясом места выполнения
```

```
кода
// Так что в результате можно получить
// Thu Jan 26 2017 11:00:00 GMT+1100 (восточно-австралийское
время)
// или
// Wed Jan 25 2017 16:00:00 GMT-0800 (тихоокеанское время)
```

`new Date(year, month, date, hours, minutes, seconds, ms)`

Создать объект `Date` с заданными компонентами в местном часовом поясе. Обязательны только первые два аргумента.

> `year` должен состоять из четырёх цифр: значение `2013` корректно, `98` – нет.
>
> `month` начинается с `0` (январь) по `11` (декабрь).
>
> Параметр `date` здесь представляет собой день месяца. Если параметр не задан, то принимается значение `1`.
>
> Если параметры `hours/minutes/seconds/ms` отсутствуют, их значением становится `0`.

Например:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // то же самое, так как часы и проч.
равны 0
```

Максимальная точность – 1 мс (до 1/1000 секунды):

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Получение компонентов даты

Существуют методы получения года, месяца и т.д. из объекта `Date`:

getFullYear()

Получить год (4 цифры)

getMonth()

Получить месяц, **от 0 до 11**.

getDate()

Получить день месяца, от 1 до 31, что несколько противоречит названию метода.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Получить, соответственно, часы, минуты, секунды или миллисекунды.

> **Никакого `getYear()`. Только `getFullYear()`**
>
> Многие интерпретаторы JavaScript реализуют нестандартный и устаревший метод `getYear()`, который порой возвращает год в виде двух цифр. Пожалуйста, обходите его стороной. Если нужно значение года, используйте `getFullYear()`.

Кроме того, можно получить определённый день недели:

getDay()

Вернуть день недели от `0` (воскресенье) до `6` (суббота). Несмотря на то, что в ряде стран за первый день недели принят понедельник, в JavaScript начало недели приходится на воскресенье.

Все вышеперечисленные методы возвращают

значения в соответствии с местным часовым поясом.

Однако существуют и их UTC-варианты, возвращающие день, месяц, год для временной зоны UTC+0: getUTCFullYear(), getUTCMonth(), getUTCDay(). Для их использования требуется после `"get"` подставить `"UTC"`.

Если ваш местный часовой пояс смещён относительно UTC, то следующий код покажет разные часы:

```
// текущая дата
let date = new Date();

// час в вашем текущем часовом поясе
alert( date.getHours() );

// час в часовом поясе UTC+0 (лондонское время без перехода на летнее время)
alert( date.getUTCHours() );
```

Помимо вышеприведённых методов, существуют два особых метода без UTC-варианта:

getTime()

Для заданной даты возвращает таймстамп – количество миллисекунд, прошедших с 1 января 1970 года UTC+0.

getTimezoneOffset()

Возвращает разницу в минутах между местным часовым поясом и UTC:

```
// если вы в часовом поясе UTC-1, то выводится 60
// если вы в часовом поясе UTC+3, выводится -180
alert( new Date().getTimezoneOffset() );
```

Установка компонентов даты

Следующие методы позволяют установить компоненты даты и времени:

- `setFullYear(year, [month], [date])`
- `setMonth(month, [date])`
- `setDate(date)`
- `setHours(hour, [min], [sec], [ms])`
- `setMinutes(min, [sec], [ms])`
- `setSeconds(sec, [ms])`
- `setMilliseconds(ms)`
- `setTime(milliseconds)` (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC)

У всех этих методов, кроме `setTime()`, есть UTC-вариант, например: `setUTCHours()`.

Как мы видим, некоторые методы могут устанавливать сразу несколько компонентов даты, например: `setHours`. Если какая-то компонента не указана, она не меняется.

Пример:

```
let today = new Date();

today.setHours(0);
alert(today); // выводится сегодняшняя дата, но значение часа будет 0

today.setHours(0, 0, 0, 0);
alert(today); // всё ещё выводится сегодняшняя дата, но время будет ровно 00:00:00.
```

Автоисправление даты

Автоисправление – это очень полезная особенность объектов `Date`. Можно устанавливать компоненты даты вне обычного диапазона значений, а объект сам себя

исправит.

Пример:

```
let date = new Date(2013, 0, ); // 32 Jan 2013 ?!?
alert(date); // ...1st Feb 2013!
```
32

Неправильные компоненты даты автоматически распределяются по остальным.

Предположим, нам требуется увеличить дату «28 февраля 2016» на два дня. В зависимости от того, високосный это год или нет, результатом будет «2 марта» или «1 марта». Нам об этом думать не нужно. Просто прибавляем два дня. Объект `Date` позаботится об остальном:

```
let date = new Date(2016, 1, 28);

alert( date ); // 1 Mar 2016
date.setDate(date.getDate() + 2);
```

Эту возможность часто используют, чтобы получить дату по прошествии заданного отрезка времени. Например, получим дату «спустя 70 секунд с текущего момента»:

```
let date = new Date();
date.setSeconds(date.getSeconds() + 70);

alert( date ); // выводит правильную дату
```

Также можно установить нулевые или даже отрицательные значения. Например:

```
let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // задать первое число месяца
alert( date );
```

```
date.setDate(0); // первый день месяца -- это 1, так что
выводится последнее число предыдущего месяца
alert( date ); // 31 Dec 2015
```

Преобразование к числу, разность дат

Если объект `Date` преобразовать в число, то получим таймстамп по аналогии с `date.getTime()`:

```
let date = new Date();
alert(+date); // количество миллисекунд, то же самое, что
date.getTime()
```

Важный побочный эффект: даты можно вычитать, в результате получаем разность в миллисекундах.

Этот приём можно использовать для измерения времени:

```
let start = new Date(); // начинаем отсчёт времени

// выполняем некоторые действия
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}

let end = new Date(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` );
```

Date.now()

Если нужно просто измерить время, объект `Date` нам не нужен.

Существует особый метод `Date.now()`, возвращающий текущую метку времени.

Семантически он эквивалентен `new Date().getTime()`, однако метод не создаёт промежуточный объект `Date`. Так что этот способ работает быстрее и не нагружает сборщик мусора.

Данный метод используется из соображений удобства или когда важно быстродействие, например, при разработке игр на JavaScript или других специализированных приложений.

Вероятно, предыдущий пример лучше переписать так:

```
// выполняем некоторые действия
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}
```

```
alert( `Цикл отработал за ${end - start} миллисекунд` ); // вычитаются числа, а не даты
let start = Date.now(); // количество миллисекунд с 1 января 1970 года
let end = Date.now(); // заканчиваем отсчёт времени
```

Бенчмаркинг

Будьте внимательны, если хотите точно протестировать производительность функции, которая зависит от процессора.

Например, сравним две функции, вычисляющие разницу между двумя датами: какая сработает быстрее?

Подобные вычисления, замеряющие производительность, также называют «бенчмарками» (benchmark).

```
// есть date1 и date2, какая функция быстрее вернёт разницу между ними в миллисекундах?
function diffSubtract(date1, date2) {
  return date2 - date1;
}

// или
```

```
function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}
```

Обе функции делают буквально одно и то же, только одна использует явный метод `date.getTime()` для получения даты в миллисекундах, а другая полагается на преобразование даты в число. Результат их работы всегда один и тот же.

Но какая функция быстрее?

Для начала можно запустить их много раз подряд и засечь разницу. В нашем случае функции очень простые, так что потребуется хотя бы 100000 повторений.

Проведём измерения:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

alert( 'Время diffSubtract: ' + bench(diffSubtract) + 'мс' );
alert( 'Время diffGetTime: ' + bench(diffGetTime) + 'мс' );
```

Вот это да! Метод `getTime()` работает ощутимо быстрее! Всё потому, что не производится преобразование типов, и интерпретаторам такое намного легче оптимизировать.

Замечательно, это уже что-то. Но до хорошего бенчмарка нам ещё далеко.

Представьте, что при выполнении `bench(diffSubtract)` процессор параллельно делал что-то ещё, также потребляющее ресурсы. А к началу выполнения `bench(diffGetTime)` он это уже завершил.

Достаточно реалистичный сценарий в современных многопроцессорных операционных системах.

В итоге у первого бенчмарка окажется меньше ресурсов процессора, чем у второго. Это может исказить результаты.

Для получения наиболее достоверных результатов тестирования производительности весь набор бенчмарков нужно запускать по нескольку раз.

Например, так:

```
function diffSubtract(date1, date2) {
  return date2 - date1;
}

function diffGetTime(date1, date2) {
  return date2.getTime() - date1.getTime();
}

function bench(f) {
  let date1 = new Date(0);
  let date2 = new Date();

  let start = Date.now();
  for (let i = 0; i < 100000; i++) f(date1, date2);
  return Date.now() - start;
}

let time1 = 0;
let time2 = 0;



alert( 'Итоговое время diffSubtract: ' + time1 );
alert( 'Итоговое время diffGetTime: ' + time2 );
// bench(upperSlice) и bench(upperLoop)
поочерёдно запускаются 10 раз
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
```

```
  time2 += bench(diffGetTime);
}
```

Современные интерпретаторы JavaScript начинают применять продвинутые оптимизации только к «горячему коду», выполняющемуся несколько раз (незачем оптимизировать то, что редко выполняется). Так что в примере выше первые запуски не оптимизированы должным образом. Нелишним будет добавить предварительный запуск для «разогрева»:

```
// добавляем для "разогрева" перед основным циклом
bench(diffSubtract);
bench(diffGetTime);

// а теперь тестируем производительность
for (let i = 0; i < 10; i++) {
  time1 += bench(diffSubtract);
  time2 += bench(diffGetTime);
}
```

Будьте осторожны с микробенчмарками

Современные интерпретаторы JavaScript выполняют множество оптимизаций. Они могут повлиять на результаты «искусственных тестов» по сравнению с «нормальным использованием», особенно если мы тестируем что-то очень маленькое, например, работу оператора или встроенной функции. Поэтому если хотите серьёзно понять производительность, пожалуйста, изучите, как работают интерпретаторы JavaScript. И тогда вам, вероятно, уже не понадобятся микробенчмарки.

Отличный набор статей о V8 можно найти на http://mrale.ph.

Разбор строки с датой

Метод Date.parse(str) считывает дату из строки.

Формат строки должен быть следующим: `YYYY-MM-DDTHH:mm:ss.sssZ`, где:

- `YYYY-MM-DD` – это дата: год-месяц-день.
- Символ `"T"` используется в качестве разделителя.
- `HH:mm:ss.sss` – время: часы, минуты, секунды и миллисекунды.
- Необязательная часть `'Z'` обозначает часовой пояс в формате `+-hh:mm`. Если указать просто букву `Z`, то получим UTC+0.

Возможны и более короткие варианты, например, `YYYY-MM-DD` или `YYYY-MM`, или даже `YYYY`.

Вызов `Date.parse(str)` обрабатывает строку в заданном формате и возвращает таймстамп (количество миллисекунд с 1 января 1970 года UTC+0). Если формат неправильный, возвращается `NaN`.

Например:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417 (таймстамп)
```

Можно тут же создать объект `new Date` из таймстампа:

```
let date = new Date(
Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

Итого

Дата и время в JavaScript представлены объектом Date. Нельзя создать «только дату» или «только время»: объекты `Date` всегда содержат и то, и другое.

Счёт месяцев начинается с нуля (да, январь – это нулевой месяц).

Дни недели в `getDay()` также отсчитываются с нуля, что соответствует воскресенью.

Объект `Date` самостоятельно корректируется при введении значений, выходящих за рамки допустимых. Это полезно для сложения/вычитания дней/месяцев/недель.

Даты можно вычитать, и разность возвращается в миллисекундах. Так происходит, потому что при преобразовании в число объект `Date` становится таймстампом.

Используйте `Date.now()` для быстрого получения текущего времени в формате таймстампа.

Учтите, что, в отличие от некоторых других систем, в JavaScript таймстамп в миллисекундах, а не в секундах.

Порой нам нужно измерить время с большей точностью. Собственными средствами JavaScript измерять время в микросекундах (одна миллионная секунды) нельзя, но в большинстве сред такая возможность есть. К примеру, в браузерах есть метод performance.now(), возвращающий количество миллисекунд с начала загрузки страницы с точностью до микросекунд (3 цифры после точки):

```
alert(`Загрузка началась ${performance.now()}мс назад`);
// Получаем что-то вроде: "Загрузка началась
34731.26000000001мс назад"
// .26 -- это микросекунды (260 микросекунд)
// корректными являются только первые три цифры после точки, а
остальные -- это ошибка точности
```

В Node.js для этого предусмотрен модуль `microtime` и ряд других способов. Технически почти любое устройство

или среда позволяет добиться большей точности, просто её нет в объекте `Date`.

Задачи

Создайте дату

важность: 5

Создайте объект `Date` для даты: 20 февраля 2012 года, 3 часа 12 минут. Временная зона – местная.

Для вывода используйте `alert`.

К решению

Покажите день недели

важность: 5

Напишите функцию `getWeekDay(date)`, показывающую день недели в коротком формате: «ПН», «ВТ», «СР», «ЧТ», «ПТ», «СБ», «ВС».

Например:

```
let date = new Date(2012, 0, 3);  // 3 января 2012 года
alert( getWeekDay(date) );        // нужно вывести "ВТ"
```

Открыть песочницу с тестами для задачи.

К решению

День недели в европейской

нумерации

важность: 5

В Европейских странах неделя начинается с понедельника (день номер 1), затем идёт вторник (номер 2) и так до воскресенья (номер 7). Напишите функцию `getLocalDay(date)`, которая возвращает «европейский» день недели для даты `date`.

```
let date = new Date(2012, 0, 3);  // 3 января 2012 года
alert( getLocalDay(date) );       // вторник, нужно показать 2
```

Открыть песочницу с тестами для задачи.

К решению

Какой день месяца был много дней назад?

важность: 4

Создайте функцию `getDateAgo(date, days)`, возвращающую число, которое было `days` дней назад от даты `date`.

К примеру, если сегодня двадцатое число, то `getDateAgo(new Date(), 1)` вернёт девятнадцатое и `getDateAgo(new Date(), 2)` – восемнадцатое.

Функция должна надёжно работать при значении `days=365` и больших значениях:

```
let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
```

```
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

P.S. Функция не должна изменять переданный ей объект `date`.

Открыть песочницу с тестами для задачи.

К решению

Последнее число месяца?

важность: 5

Напишите функцию `getLastDayOfMonth(year, month)`, возвращающую последнее число месяца. Иногда это 30, 31 или даже февральские 28/29.

Параметры:

- `year` – год из четырёх цифр, например, 2012.
- `month` – месяц от 0 до 11.

К примеру, `getLastDayOfMonth(2012, 1) = 29` (високосный год, февраль).

Открыть песочницу с тестами для задачи.

К решению

Сколько сегодня прошло секунд?

важность: 5

Напишите функцию `getSecondsToday()`,

возвращающую количество секунд с начала сегодняшнего дня.

Например, если сейчас `10:00`, и не было перехода на зимнее/летнее время, то:

```
getSecondsToday() == 36000 // (3600 * 10)
```

Функция должна работать в любой день, т.е. в ней не должно быть конкретного значения сегодняшней даты.

К решению

Сколько секунд осталось до завтра?

важность: 5

Создайте функцию `getSecondsToTomorrow()`, возвращающую количество секунд до завтрашней даты.

Например, если сейчас `23:00`, то:

```
getSecondsToTomorrow() == 3600
```

P.S. Функция должна работать в любой день, т.е. в ней не должно быть конкретного значения сегодняшней даты.

К решению

Форматирование относительной даты

важность: 4

Напишите функцию `formatDate(date)`, форматирующую `date` по следующему принципу:

- Если спустя `date` прошло менее 1 секунды, вывести `"прямо сейчас"`.
- В противном случае, если с `date` прошло меньше 1 минуты, вывести `"n сек. назад"`.
- В противном случае, если меньше часа, вывести `"m мин. назад"`.
- В противном случае, полная дата в формате `"DD.MM.YY HH:mm"`. А именно: `"день.месяц.год часы:минуты"`, всё в виде двух цифр, т.е. `31.12.16 10:00`.

Например:

```
alert( formatDate(new Date(new Date - 1)) ); // "прямо сейчас"

alert( formatDate(new Date(new Date - 30 * 1000)) ); // "30 сек. назад"

alert( formatDate(new Date(new Date - 5 * 60 * 1000)) ); // "5 мин. назад"

// вчерашняя дата вроде 31.12.2016, 20:00
alert( formatDate(new Date(new Date - 86400 * 1000)) );
```

Открыть песочницу с тестами для задачи.

К решению

Формат JSON, метод toJSON

Допустим, у нас есть сложный объект, и мы хотели бы преобразовать его в строку, чтобы отправить по сети или просто вывести для логирования.

Естественно, такая строка должна включать в себя все важные свойства.

Мы могли бы реализовать преобразование следующим образом:

```
let user = {
  name: "John",
  age: 30,


};

alert(user); // {name: "John", age: 30}
  toString() {
    return `{name: "${this.name}", age:
${this.age}}`;
  }
```

…Но в процессе разработки добавляются новые свойства, старые свойства переименовываются и удаляются. Обновление такого `toString` каждый раз может стать проблемой. Мы могли бы попытаться перебрать свойства в нём, но что, если объект сложный, и в его свойствах имеются вложенные объекты? Мы должны были бы осуществить их преобразование тоже.

К счастью, нет необходимости писать код для обработки всего этого. У задачи есть простое решение.

JSON.stringify

JSON (JavaScript Object Notation) – это общий формат для представления значений и объектов. Его описание задокументировано в стандарте RFC 4627. Первоначально

он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним. Таким образом, JSON легко использовать для обмена данными, когда клиент использует JavaScript, а сервер написан на Ruby/PHP/Java или любом другом языке.

JavaScript предоставляет методы:

- `JSON.stringify` для преобразования объектов в JSON.
- `JSON.parse` для преобразования JSON обратно в объект.

Например, здесь мы преобразуем через `JSON.stringify` данные студента:

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};



alert(typeof json); // мы получили строку!

alert(json);
let json = JSON.stringify(student);
/* выведет объект в формате JSON:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"],
  "wife": null
}
*/
```

Метод `JSON.stringify(student)` берёт объект и

преобразует его в строку.

Полученная строка `json` называется *JSON-форматированным* или *сериализованным* объектом. Мы можем отправить его по сети или поместить в обычное хранилище данных.

Обратите внимание, что объект в формате JSON имеет несколько важных отличий от объектного литерала:

- Строки используют двойные кавычки. Никаких одинарных кавычек или обратных кавычек в JSON. Так `'John'` становится `"John"`.
- Имена свойств объекта также заключаются в двойные кавычки. Это обязательно. Так `age:30` становится `"age":30`.

`JSON.stringify` может быть применён и к примитивам.

JSON поддерживает следующие типы данных:

- Объекты `{ ... }`
- Массивы `[ ... ]`
- Примитивы:
  - строки,
  - числа,
  - логические значения `true/false`,
  - `null`.

Например:

```
// число в JSON остаётся числом
alert( JSON.stringify(1) ) // 1

// строка в JSON по-прежнему остаётся строкой, но в двойных кавычках
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true
```

```
alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON является независимой от языка спецификацией для данных, поэтому `JSON.stringify` пропускает некоторые специфические свойства объектов JavaScript.

А именно:

- Свойства-функции (методы).
- Символьные свойства.
- Свойства, содержащие `undefined`.

```
let user = {
  sayHi() { // будет пропущено
    alert("Hello");
  },
  [Symbol("id")]: 123, // также будет пропущено
  something: undefined // как и это - пропущено
};

alert( JSON.stringify(user) ); // {} (пустой объект)
```

Обычно это нормально. Если это не то, чего мы хотим, то скоро мы увидим, как можно настроить этот процесс.

Самое замечательное, что вложенные объекты поддерживаются и конвертируются автоматически.

Например:

```
let meetup = {
  title: "Conference",

};

alert( JSON.stringify(meetup) );
/* вся структура преобразована в строку:
{
  "title":"Conference",
  "room":{"number":23,"participants":["john","ann"]},
}
*/
```

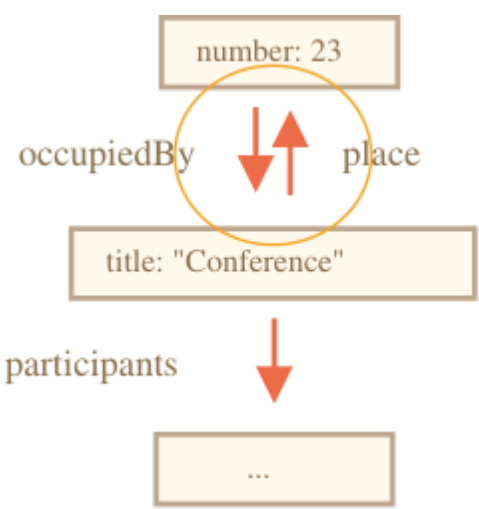
```
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
```

Важное ограничение: не должно быть циклических ссылок.

Например:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: ["john", "ann"]
};

meetup.place = room;       // meetup ссылается на room
room.occupiedBy = meetup; // room ссылается на meetup

JSON.stringify(meetup); // Ошибка:
Преобразование цикличной структуры в JSON
```

Здесь преобразование завершается неудачно из-за циклической ссылки: `room.occupiedBy` ссылается на `meetup`, и `meetup.place` ссылается на `room`:



Исключаем и преобразуем: replacer

Полный синтаксис `JSON.stringify`:

```
let json = JSON.stringify(value[, replacer, space])
```

value

Значение для кодирования.

replacer

Массив свойств для кодирования или функция соответствия `function(key, value)`.

space

Дополнительное пространство (отступы), используемое для форматирования.

В большинстве случаев `JSON.stringify` используется только с первым аргументом. Но если нам нужно настроить процесс замены, например, отфильтровать циклические ссылки, то можно использовать второй аргумент `JSON.stringify`.

Если мы передадим ему массив свойств, будут закодированы только эти свойства.

Например:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup ссылается на room
};
```

```
room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ) );
// {"title":"Conference","participants":[{},{}]}
[
'title', 'participants']
```

Здесь мы, наверное, слишком строги. Список свойств применяется ко всей структуре объекта. Так что внутри `participants` – пустые объекты, потому что `name` нет в списке.

Давайте включим в список все свойства, кроме `room.occupiedBy`, из-за которого появляется цикличная ссылка:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup ссылается на room
};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ) );
/*
{
  "title":"Conference",
  "participants":[{"name":"John"},{"name":"Alice"}],
  "place":{"number":23}
}
*/
[
'title', 'participants', 'place', 'name',
'number']
```

Теперь всё, кроме `occupiedBy`, сериализовано. Но список свойств довольно длинный.

К счастью, в качестве `replacer` мы можем использовать функцию, а не массив.

Функция будет вызываться для каждой пары `(key, value)`, и она должна возвращать заменённое значение, которое будет использоваться вместо исходного. Или `undefined`, чтобы пропустить значение.

В нашем случае мы можем вернуть `value` «как есть» для всего, кроме `occupiedBy`. Чтобы игнорировать `occupiedBy`, код ниже возвращает `undefined`:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  participants: [{name: "John"}, {name: "Alice"}],
  place: room // meetup ссылается на room
};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, function replacer(key, value) {
  alert(`${key}: ${value}`);
  return (key == 'occupiedBy') ? undefined : value;
}));

/* пары ключ:значение, которые приходят в replacer:
:             [object Object]
title:        Conference
participants: [object Object],[object Object]
0:            [object Object]
name:         John
1:            [object Object]
name:         Alice
place:        [object Object]
number:       23
*/
```

Обратите внимание, что функция `replacer` получает каждую пару ключ/значение, включая вложенные объекты и элементы массива. И она применяется рекурсивно. Значение `this` внутри `replacer` – это объект, который содержит текущее свойство.

Первый вызов – особенный. Ему передаётся специальный «объект-обёртка»: `{"": meetup}`. Другими словами, первая `(key, value)` пара имеет пустой ключ, а значением является целевой объект в общем. Вот почему первая строка из примера выше будет `":[object Object]"`.

Идея состоит в том, чтобы дать как можно больше возможностей `replacer` – у него есть возможность проанализировать и заменить/пропустить даже весь объект целиком, если это необходимо.

Форматирование: space

Третий аргумент в `JSON.stringify(value, replacer, space)` – это количество пробелов, используемых для удобного форматирования.

Ранее все JSON-форматированные объекты не имели отступов и лишних пробелов. Это нормально, если мы хотим отправить объект по сети. Аргумент `space` используется исключительно для вывода в удобочитаемом виде.

Ниже `space = 2` указывает JavaScript отображать вложенные объекты в несколько строк с отступом в 2 пробела внутри объекта:

```
let user = {
  name: "John",
  age: 25,
  roles: {
    isAdmin: false,
    isEditor: true
  }
};

alert(JSON.stringify(user, null, 2));
/* отступ в 2 пробела:
{
  "name": "John",
```

```
  "age": 25,
  "roles": {
    "isAdmin": false,
    "isEditor": true
  }
}
*/

/* для JSON.stringify(user, null, 4) результат содержит больше
отступов:
{
    "name": "John",
    "age": 25,
    "roles": {
        "isAdmin": false,
        "isEditor": true
    }
}
*/
```

Параметр `space` применяется для логирования и красивого вывода.

Пользовательский «toJSON»

Как и `toString` для преобразования строк, объект может предоставлять метод `toJSON` для преобразования в JSON. `JSON.stringify` автоматически вызывает его, если он есть.

Например:

```
let room = {
  number: 23
};

let meetup = {
  title: "Conference",
  date: new Date(Date.UTC(2017, 0, 1)),
  room
};

alert( JSON.stringify(meetup) );
/*
  {
```

```
    "title":"Conference",

    "room": {"number":23}              // (2)
  }
*/
     "date":"2017-01-01T00:00:00.000Z",  // (1)
```

Как видим, `date (1)` стал строкой. Это потому, что все объекты типа `Date` имеют встроенный метод `toJSON`, который возвращает такую строку.

Теперь давайте добавим собственную реализацию метода `toJSON` в наш объект `room (2)`:

```
let room = {
  number: 23,

};

let meetup = {
  title: "Conference",
  room
};



alert( JSON.stringify(meetup) );
/*
  {
    "title":"Conference",

  }
*/
   toJSON() {
     return this.number;
   }
alert( JSON.stringify(room) ); // 23
     "room": 23
```

Как видите, `toJSON` используется как при прямом вызове `JSON.stringify(room)`, так и когда `room` вложен в другой сериализуемый объект.

JSON.parse

Чтобы декодировать JSON-строку, нам нужен другой метод с именем JSON.parse.

Синтаксис:

```
let value = JSON.parse(str, [reviver]);
```

str

JSON для преобразования в объект.

reviver

Необязательная функция, которая будет вызываться для каждой пары `(ключ, значение)` и может преобразовывать значение.

Например:

```
// строковый массив
let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1
```

Или для вложенных объектов:

```
let user = '{ "name": "John", "age": 35, "isAdmin": false,
"friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1
```

JSON может быть настолько сложным, насколько это необходимо, объекты и массивы могут включать другие

объекты и массивы. Но они должны быть в том же JSON-формате.

Вот типичные ошибки в написанном от руки JSON (иногда приходится писать его для отладки):

```
let json = `{
  : "John",                     // Ошибка: имя свойства без кавычек
  "surname": ,                  // Ошибка: одинарные кавычки в значении (должны быть двойными)
  : false                   // Ошибка: одинарные кавычки в ключе (должны быть двойными)
  "birthday": , // Ошибка: не допускается конструктор "new", только значения.
  "friends": [0,1,2,3]                      // Здесь всё в порядке
}`;
name
'Smith'
'isAdmin'
new Date(2000, 2, 3)
```

Кроме того, JSON не поддерживает комментарии. Добавление комментария в JSON делает его недействительным.

Существует ещё один формат JSON5, который поддерживает ключи без кавычек, комментарии и т.д. Но это самостоятельная библиотека, а не спецификация языка.

Обычный JSON настолько строг не потому, что его разработчики ленивы, а потому, что позволяет легко, надёжно и очень быстро реализовывать алгоритм кодирования и чтения.

Использование reviver

Представьте, что мы получили объект `meetup` с сервера в

виде строки данных.

Вот такой:

```
// title: (meetup title), date: (meetup date)
let str =
'{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

…А теперь нам нужно *десериализовать* её, т.е. снова превратить в объект JavaScript.

Давайте сделаем это, вызвав `JSON.parse`:

```
let str =
'{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

Ой, ошибка!

Значением `meetup.date` является строка, а не `Date` объект. Как `JSON.parse` мог знать, что он должен был преобразовать эту строку в `Date`?

Давайте передадим `JSON.parse` функцию восстановления вторым аргументом, которая возвращает все значения «как есть», но `date` станет `Date`:

```
let str =
'{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';


alert( meetup.date.getDate() ); // 30 - теперь работает!
let meetup = JSON.parse(str, function(key,
value) {
  if (key == 'date') return new Date(value);
  return value;
});
```

Кстати, это работает и для вложенных объектов:

```
let schedule = `{
  "meetups": [
    {"title":"Conference","date":"2017-11-30T12:00:00.000Z"},
    {"title":"Birthday","date":"2017-04-18T12:00:00.000Z"}
  ]
}`;

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.meetups[1].date.getDate() );
// 18 – отлично!
```

Итого

JSON – это формат данных, который имеет собственный независимый стандарт и библиотеки для большинства языков программирования.

JSON поддерживает простые объекты, массивы, строки, числа, логические значения и `null`.

JavaScript предоставляет методы JSON.stringify для сериализации в JSON и JSON.parse для чтения из JSON.

Оба метода поддерживают функции преобразования для интеллектуального чтения/записи.

Если объект имеет метод `toJSON`, то он вызывается через `JSON.stringify`.

Задачи

Преобразуйте объект в JSON, а затем обратно в обычный объект

важность: 5

Преобразуйте `user` в JSON, затем прочитайте этот JSON в другую переменную.

```
let user = {
  name: "Василий Иванович",
  age: 35
};
```

К решению

Исключить обратные ссылки

важность: 5

В простых случаях циклических ссылок мы можем исключить свойство, из-за которого они возникают, из сериализации по его имени.

Но иногда мы не можем использовать имя, так как могут быть и другие, нужные, свойства с этим именем во вложенных объектах. Поэтому можно проверять свойство по значению.

Напишите функцию `replacer` для JSON-преобразования, которая удалит свойства, ссылающиеся на `meetup`:

```
let room = {
  number: 23
};

let meetup = {
  title: "Совещание",
  occupiedBy: [{name: "Иванов"}, {name: "Петров"}],
  place: room
};
```

```
alert( JSON.stringify(meetup, function replacer(key, value) {
  /* ваш код */
}));

/* в результате должно быть:
{
  "title":"Совещание",
  "occupiedBy":[{"name":"Иванов"},{"name":"Петров"}],
  "place":{"number":23}
}
*/
// цикличные ссылки
room.occupiedBy = meetup;
meetup.self = meetup;
```

К решению

Продвинутая работа с функциями

Рекурсия и стек

Вернёмся к функциям и изучим их более подробно.

Нашей первой темой будет *рекурсия*.

Если вы не новичок в программировании, то, возможно, уже знакомы с рекурсией и можете пропустить эту главу.

Рекурсия – это приём программирования, полезный в ситуациях, когда задача может быть естественно разделена на несколько аналогичных, но более простых задач. Или когда задача может быть упрощена до несложных действий плюс простой вариант той же задачи. Или, как мы скоро увидим, для работы с определёнными структурами данных.

В процессе выполнения задачи в теле функции могут быть вызваны другие функции для выполнения подзадач. Частный случай подвызова – когда функция вызывает *сама себя*. Это как раз и называется *рекурсией*.

Два способа мышления

В качестве первого примера напишем функцию `pow(x, n)`, которая возводит `x` в натуральную степень `n`. Иначе говоря, умножает `x` на само себя `n` раз.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

Рассмотрим два способа её реализации.

1. Итеративный способ: цикл `for`:

```
function pow(x, n) {
  let result = 1;

  // умножаем result на x n раз в цикле
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

alert( pow(2, 3) ); // 8
```

2. Рекурсивный способ: упрощение задачи и вызов функцией самой себя:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```

Обратите внимание, что рекурсивный вариант отличается принципиально.

Когда функция `pow(x, n)` вызывается, исполнение делится на две ветви:

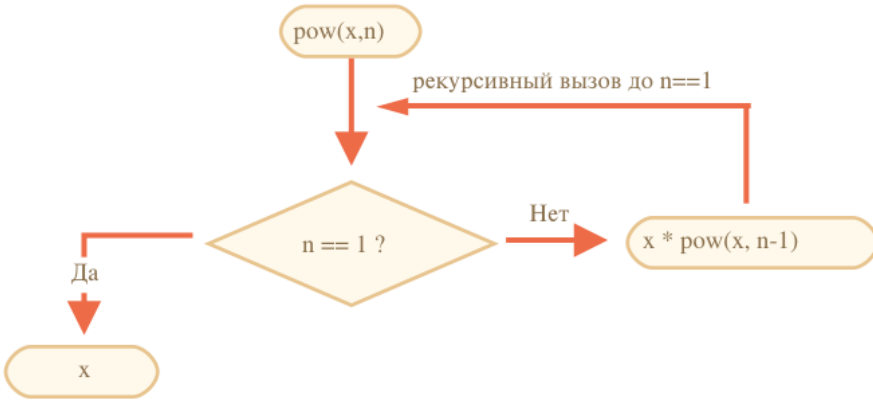
```
              if n==1  = x
             /
pow(x, n) =
             \
              else     = x * pow(x, n - 1)
```

1. Если `n == 1`, тогда всё просто. Эта ветвь называется

базой рекурсии, потому что сразу же приводит к очевидному результату: `pow(x, 1)` равно `x`.

2. Мы можем представить `pow(x, n)` в виде: `x * pow(x, n - 1)`. Что в математике записывается как: $x^n = x * x^{n-1}$. Эта ветвь – *шаг рекурсии*: мы сводим задачу к более простому действию (умножение на `x`) и более простой аналогичной задаче (`pow` с меньшим `n`). Последующие шаги упрощают задачу всё больше и больше, пока `n` не достигает `1`.

Говорят, что функция `pow` *рекурсивно вызывает саму себя* до `n == 1`.



Например, рекурсивный вариант вычисления `pow(2, 4)` состоит из шагов:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

Итак, рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – к ещё более простому и так далее, пока значение не станет очевидно.

Рекурсивное решение обычно короче

Рекурсивное решение задачи обычно короче, чем итеративное.

Используя условный оператор `?` вместо `if`, мы можем переписать `pow(x, n)`, делая код функции более лаконичным, но всё ещё легко читаемым:

```
function pow(x, n) {
  return (n == 1) ? x : (x * pow(x, n - 1));
}
```

Общее количество вложенных вызовов (включая первый) называют *глубиной рекурсии*. В нашем случае она будет равна ровно `n`.

Максимальная глубина рекурсии ограничена движком JavaScript. Точно можно рассчитывать на 10000 вложенных вызовов, некоторые интерпретаторы допускают и больше, но для большинства из них 100000 вызовов – за пределами возможностей. Существуют автоматические оптимизации, помогающие избежать переполнения стека вызовов («оптимизация хвостовой рекурсии»), но они ещё не поддерживаются везде и работают только для простых случаев.

Это ограничивает применение рекурсии, но она всё равно широко распространена: для решения большого числа задач рекурсивный способ решения даёт более простой код, который легче поддерживать.

Контекст выполнения, стек

Теперь мы посмотрим, как работают рекурсивные вызовы. Для этого заглянем «под капот» функций.

Информация о процессе выполнения запущенной функции хранится в её *контексте выполнения* (execution context).

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение `this` (мы не используем его в данном примере) и прочую служебную информацию.

Один вызов функции имеет ровно один контекст выполнения, связанный с ним.

Когда функция производит вложенный вызов, происходит следующее:

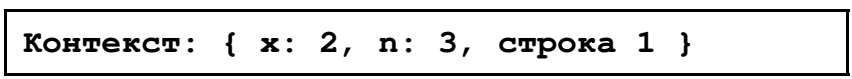
- Выполнение текущей функции приостанавливается.
- Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – *стеке контекстов выполнения*.
- Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
- После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

Разберёмся с контекстами более подробно на примере вызова функции `pow(2, 3)`.

pow(2, 3)

В начале вызова `pow(2, 3)` контекст выполнения будет хранить переменные: `x = 2, n = 3`, выполнение находится на первой строке функции.

Можно схематически изобразить это так:



pow(2, 3)

Это только начало выполнения функции. Условие `n == 1` ложно, поэтому выполнение идёт во вторую ветку `if`:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {

  }
}

alert( pow(2, 3) );
```

```
    return x * pow(x, n - 1);
```

Значения переменных те же самые, но выполнение функции перешло к другой строке, актуальный контекст:

Контекст: { x: 2, n: 3, строка 5 }

pow(2, 3)

Чтобы вычислить выражение `x * pow(x, n - 1)`, требуется произвести запуск `pow` с новыми аргументами `pow(2, 2)`.

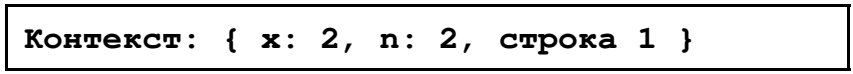
pow(2, 2)

Для выполнения вложенного вызова JavaScript запоминает текущий контекст выполнения в *стеке контекстов выполнения*.

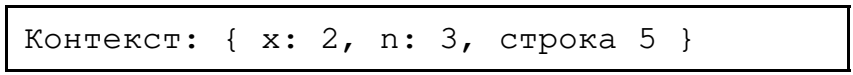
Здесь мы вызываем ту же функцию `pow`, однако это абсолютно неважно. Для любых функций процесс одинаков:

1. Текущий контекст «запоминается» на вершине стека.
2. Создаётся новый контекст для вложенного вызова.
3. Когда выполнение вложенного вызова заканчивается – контекст предыдущего вызова восстанавливается, и выполнение соответствующей функции продолжается.

Вид контекста в начале выполнения вложенного вызова `pow(2, 2)`:



pow(2, 2)



pow(2, 3)

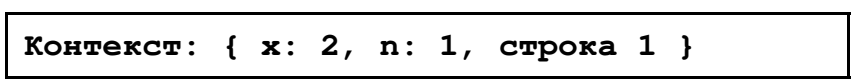
Новый контекст выполнения находится на вершине стека (и выделен жирным), а предыдущие запомненные контексты – под ним.

Когда выполнение подвызова закончится, можно будет легко вернуться назад, потому что контекст сохраняет как переменные, так и точное место кода, в котором он остановился. Слово «строка» на рисунках условно, на самом деле запоминается более точное место в цепочке команд.

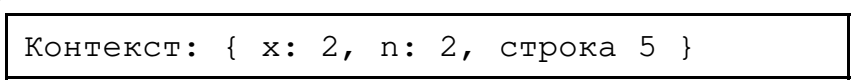
pow(2, 1)

Процесс повторяется: производится новый вызов в строке `5`, теперь с аргументами `x=2, n=1`.

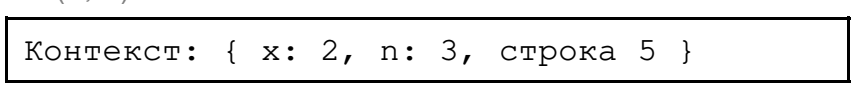
Создаётся новый контекст выполнения, предыдущий контекст добавляется в стек:



pow(2, 1)



pow(2, 2)



pow(2, 3)

Теперь в стеке два старых контекста и один текущий для `pow(2, 1)`.

Выход

При выполнении `pow(2, 1)`, в отличие от предыдущих запусков, условие `n == 1` истинно, поэтому выполняется первая ветка условия `if`:

```
function pow(x, n) {
  if (n == 1) {

  } else {
    return x * pow(x, n - 1);
  }
}
```

```
    return x;
```

Вложенных вызовов больше нет, поэтому функция завершается, возвращая `2`.

Когда функция заканчивается, контекст её выполнения больше не нужен, поэтому он удаляется из памяти, а из стека восстанавливается предыдущий:

Контекст: { x: 2, n: 2, строка 5 }

pow(2, 2)

Контекст: { x: 2, n: 3, строка 5 }

pow(2, 3)

Возобновляется обработка вызова `pow(2, 2)`. Имея результат `pow(2, 1)`, он может закончить свою работу `x * pow(x, n - 1)`, вернув `4`.

Восстанавливается контекст предыдущего вызова:

Контекст: { x: 2, n: 3, строка 5 }

pow(2, 3)

Самый внешний вызов заканчивает свою работу, его результат: `pow(2, 3) = 8`.

Глубина рекурсии в данном случае составила **3**.

Как видно из иллюстраций выше, глубина рекурсии равна максимальному числу контекстов, одновременно хранимых в стеке.

Обратим внимание на требования к памяти. Рекурсия приводит к хранению всех данных для неоконченных внешних вызовов в стеке, и в данном случае это приводит к тому, что возведение в степень `n` хранит в памяти `n` различных контекстов.

Реализация возведения в степень через цикл гораздо более экономна:

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Итеративный вариант функции `pow` использует один контекст, в котором будут последовательно меняться значения `i` и `result`. При этом объём затрачиваемой памяти небольшой, фиксированный и не зависит от `n`.

Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.

Но переделка рекурсии в цикл может быть нетривиальной, особенно когда в функции в зависимости от условий используются различные рекурсивные подвызовы, результаты которых объединяются, или когда ветвление более сложное. Оптимизация может быть ненужной и совершенно нестоящей усилий.

Часто код с использованием рекурсии более короткий, лёгкий для понимания и поддержки. Оптимизация

требуется не везде, как правило, нам важен хороший код, поэтому она и используется.

Рекурсивные обходы

Другим отличным применением рекурсии является рекурсивный обход.

Представьте, у нас есть компания. Структура персонала может быть представлена как объект:

```
let company = {
  sales: [{
    name: 'John',
    salary: 1000
  }, {
    name: 'Alice',
    salary: 600
  }],

  development: {
    sites: [{
      name: 'Peter',
      salary: 2000
    }, {
      name: 'Alex',
      salary: 1800
    }],

    internals: [{
      name: 'Jack',
      salary: 1300
    }]
  }
};
```

Другими словами, в компании есть отделы.

> Отдел может состоять из массива работников. Например, в отделе `sales` работают 2 сотрудника: Джон и Алиса.
>
> Или отдел может быть разделён на подотделы,

например, отдел `development` состоит из подотделов: `sites` и `internals`. В каждом подотделе есть свой персонал.

Также возможно, что при росте подотдела он делится на подразделения (или команды).

Например, подотдел `sites` в будущем может быть разделён на команды `siteA` и `siteB`. И потенциально они могут быть разделены ещё. Этого нет на картинке, просто нужно иметь это в виду.

Теперь, допустим, нам нужна функция для получения суммы всех зарплат. Как мы можем это сделать?

Итеративный подход не прост, потому что структура довольно сложная. Первая идея заключается в том, чтобы сделать цикл `for` поверх объекта `company` с вложенным циклом над отделами 1-го уровня вложенности. Но затем нам нужно больше вложенных циклов для итераций над сотрудниками отделов второго уровня, таких как `sites`... А затем ещё один цикл по отделам 3-го уровня, которые могут появиться в будущем? Если мы поместим в код 3-4 вложенных цикла для обхода одного объекта, то это будет довольно некрасиво.

Давайте попробуем рекурсию.

Как мы видим, когда наша функция получает отдел для подсчёта суммы зарплат, есть два возможных случая:

1. Либо это «простой» отдел с *массивом* – тогда мы сможем суммировать зарплаты в простом цикле.
2. Или это *объект* с `N` подотделами – тогда мы можем сделать `N` рекурсивных вызовов, чтобы получить сумму для каждого из подотделов, и объединить результаты.

Случай (1), когда мы получили массив, является базой

рекурсии, тривиальным случаем.

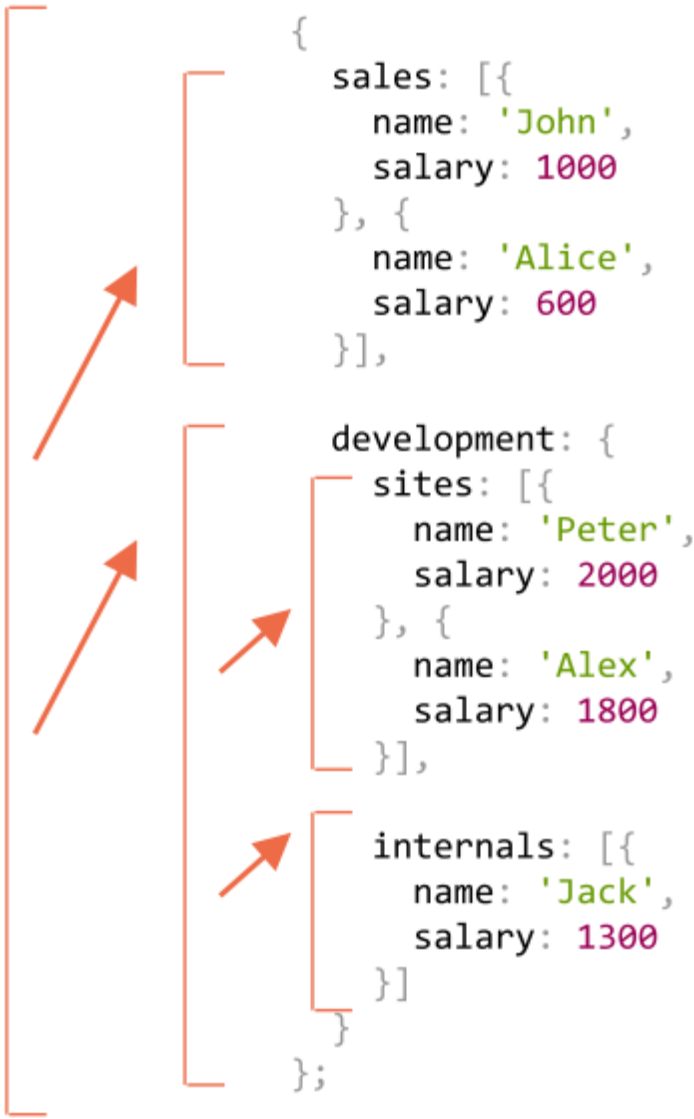
Случай (2), при получении объекта, является шагом рекурсии. Сложная задача разделяется на подзадачи для подотделов. Они могут, в свою очередь, снова разделиться на подотделы, но рано или поздно это разделение закончится, и решение сведётся к случаю (1).

Алгоритм даже проще читается в виде кода:

```
let company = { // тот же самый объект, сжатый для краткости
  sales: [{name: 'John', salary: 1000}, {name: 'Alice',
salary: 600 }],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex',
salary: 1800 }],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

// Функция для подсчёта суммы зарплат


alert(sumSalaries(company)); // 6700
function sumSalaries(department) {
  if (Array.isArray(department)) { // случай
(1)
    return department.reduce((prev, current)
=> prev + current.salary, 0); // сумма
элементов массива
  } else { // случай (2)
    let sum = 0;
    for (let subdep of
Object.values(department)) {
      sum += sumSalaries(subdep); //
рекурсивно вызывается для подотделов, суммируя
результаты
    }
    return sum;
  }
}
```

Код краток и прост для понимания (надеюсь?). В этом сила рекурсии. Она работает на любом уровне вложенности отделов.

Схема вызовов:



Принцип прост: для объекта `{...}` используются рекурсивные вызовы, а массивы `[...]` являются «листьями» дерева рекурсии, они сразу дают результат.

Обратите внимание, что в коде используются возможности, о которых мы говорили ранее:

- Метод `arr.reduce` из главы Методы массивов для получения суммы элементов массива.
- Цикл `for(val of Object.values(obj))` для итерации по значениям объекта: `Object.values` возвращает массив значений.

Рекурсивные структуры

Рекурсивная (рекурсивно определяемая) структура данных – это структура, которая повторяет саму себя в своих частях.

Мы только что видели это на примере структуры компании выше.

Отдел компании – это:

- Либо массив людей.
- Либо объект с *отделами*.

Для веб-разработчиков существуют гораздо более известные примеры: HTML- и XML-документы.

В HTML-документе *HTML-тег* может содержать:

- Фрагменты текста.
- HTML-комментарии.
- Другие *HTML-теги* (которые, в свою очередь, могут содержать фрагменты текста/комментарии или другие теги и т.д.).

Это снова рекурсивное определение.

Для лучшего понимания мы рассмотрим ещё одну

рекурсивную структуру под названием «связанный список», которая в некоторых случаях может использоваться в качестве альтернативы массиву.

Связанный список

Представьте себе, что мы хотим хранить упорядоченный список объектов.

Естественным выбором будет массив:

```
let arr = [obj1, obj2, obj3];
```

…Но у массивов есть недостатки. Операции «удалить элемент» и «вставить элемент» являются дорогостоящими. Например, операция `arr.unshift(obj)` должна переиндексировать все элементы, чтобы освободить место для нового `obj`, и, если массив большой, на это потребуется время. То же самое с `arr.shift()`.

Единственные структурные изменения, не требующие массовой переиндексации – это изменения, которые выполняются с конца массива: `arr.push/pop`. Таким образом, массив может быть довольно медленным для больших очередей, когда нам приходится работать с его началом.

Или же, если нам действительно нужны быстрые вставка/удаление, мы можем выбрать другую структуру данных, называемую связанный список.

Элемент *связанного списка* определяется рекурсивно как объект с:

- `value`,
- `next` – свойство, ссылающееся на следующий *элемент связанного списка* или `null`, если это последний элемент.

Пример:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

Графическое представление списка:

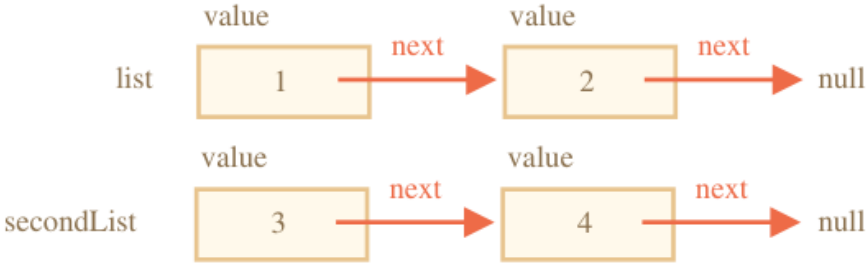
Альтернативный код для создания:

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };
```

Здесь мы можем ещё лучше увидеть, что есть несколько объектов, каждый из которых имеет `value` и `next`, указывающий на соседа. Переменная `list` является первым объектом в цепочке, поэтому, следуя по указателям `next` из неё, мы можем попасть в любой элемент.

Список можно легко разделить на несколько частей и впоследствии объединить обратно:

```
let secondList = list.next.next;
list.next.next = null;
```



Для объединения:

```
list.next.next = secondList;
```

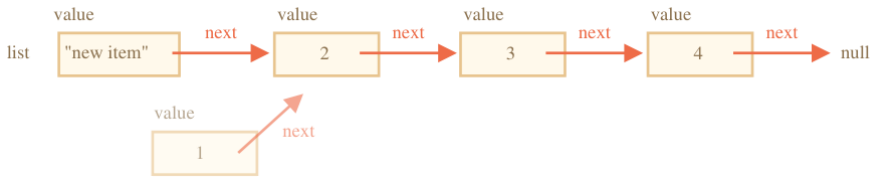
И, конечно, мы можем вставить или удалить элементы из любого места.

Например, для добавления нового элемента нам нужно обновить первый элемент списка:

```
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// добавление нового элемента в список
list = { value: "new item", next: list };
```

Чтобы удалить элемент из середины списка, нужно изменить значение `next` предыдущего элемента:

```
list.next = list.next.next;
```



`list.next` перепрыгнуло с `1` на значение `2`. Значение `1` теперь исключено из цепочки. Если оно не хранится где-нибудь ещё, оно будет автоматически удалено из памяти.

В отличие от массивов, нет перенумерации, элементы легко переставляются.

Естественно, списки не всегда лучше массивов. В противном случае все пользовались бы только списками.

Главным недостатком является то, что мы не можем легко получить доступ к элементу по его индексу. В простом массиве: `arr[n]` является прямой ссылкой. Но в списке мы должны начать с первого элемента и перейти в `next` N раз, чтобы получить N-й элемент.

…Но нам не всегда нужны такие операции. Например, нам может быть нужна очередь или даже двухсторонняя очередь – это упорядоченная структура, которая позволяет очень быстро добавлять/удалять элементы с обоих концов, но там не нужен доступ в середину.

Списки могут быть улучшены:

- Можно добавить свойство `prev` в дополнение к `next` для ссылки на предыдущий элемент, чтобы легко двигаться по списку назад.

  Можно также добавить переменную `tail`, которая будет ссылаться на последний элемент списка (и обновлять её при добавлении/удалении элементов с конца).

  …Возможны другие изменения: главное, чтобы структура данных соответствовала нашим задачам с точки зрения производительности и удобства.

Итого

Термины:

Рекурсия – это термин в программировании, означающий вызов функцией самой себя. Рекурсивные функции могут быть использованы для элегантного решения определённых задач.

Когда функция вызывает саму себя, это называется *шагом рекурсии*. *База* рекурсии – это такие аргументы функции, которые делают задачу настолько простой, что решение не требует дальнейших вложенных вызовов.

Рекурсивно определяемая структура данных – это структура данных, которая может быть определена с использованием самой себя.

Например, связанный список может быть определён как структура данных, состоящая из объекта, содержащего ссылку на список (или null).

```
list = { value, next -> list }
```

Деревья, такие как дерево HTML-элементов или дерево отделов из этой главы, также являются рекурсивными: они разветвляются, и каждая ветвь может содержать другие ветви.

Как мы видели в примере `sumSalary`, рекурсивные функции могут быть использованы для прохода по ним.

Любая рекурсивная функция может быть переписана в итеративную. И это иногда требуется для оптимизации работы. Но для многих задач рекурсивное решение достаточно быстрое и простое в написании и поддержке.

Задачи

Вычислить сумму чисел до данного

важность: 5

Напишите функцию `sumTo(n)`, которая вычисляет сумму чисел `1 + 2 + ... + n`.

Например:

```
sumTo(1) = 1
sumTo(2) = 2 + 1 = 3
sumTo(3) = 3 + 2 + 1 = 6
sumTo(4) = 4 + 3 + 2 + 1 = 10
...
sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050
```

Сделайте три варианта решения:

1. С использованием цикла.
2. Через рекурсию, т.к. `sumTo(n) = n + sumTo(n-1)` for `n > 1`.
3. С использованием формулы арифметической прогрессии.

Пример работы вашей функции:

```
function sumTo(n) { /*... ваш код ... */ }

alert( sumTo(100) ); // 5050
```

P.S. Какой вариант решения самый быстрый? Самый медленный? Почему?

P.P.S. Можно ли при помощи рекурсии посчитать `sumTo(100000)`?

К решению

Вычислить факториал

важность: 4

Факториал натурального числа – это число, умноженное на `"себя минус один"`, затем на `"себя минус два"`, и так далее до `1`. Факториал `n` обозначается как `n!`

Определение факториала можно записать как:

```
n! = n * (n - 1) * (n - 2) * ...*1
```

Примеры значений для разных `n`:

```
1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
```

Задача – написать функцию `factorial(n)`, которая возвращает `n!`, используя рекурсию.

```
alert( factorial(5) ); // 120
```

P.S. Подсказка: `n!` можно записать как `n * (n-1)!`
Например: `3! = 3*2! = 3*2*1! = 6`

К решению

Числа Фибоначчи

важность: 5

Последовательность чисел Фибоначчи определяется формулой $F_n = F_{n-1} + F_{n-2}$. То есть, следующее число получается как сумма двух предыдущих.

Первые два числа равны `1`, затем `2(1+1)`, затем `3(1+2)`, `5(2+3)` и так далее: `1, 1, 2, 3, 5, 8, 13, 21...`.

Числа Фибоначчи тесно связаны с золотым сечением и множеством природных явлений вокруг нас.

Напишите функцию `fib(n)` которая возвращает `n-e` число Фибоначчи.

Пример работы:

```
function fib(n) { /* ваш код */ }

alert(fib(3)); // 2
alert(fib(7)); // 13
alert(fib(77)); // 5527939700884757
```

P.S. Все запуски функций из примера выше должны работать быстро. Вызов `fib(77)` должен занимать не более доли секунды.

К решению

Вывод односвязного списка

важность: 5

Допустим, у нас есть односвязный список (как описано в главе Рекурсия и стек):

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};
```

Напишите функцию `printList(list)`, которая выводит элементы списка по одному.

Сделайте два варианта решения: используя цикл и через рекурсию.

Как лучше: с рекурсией или без?

К решению

Вывод односвязного списка в обратном порядке

важность: 5

Выведите односвязный список из предыдущего задания Вывод односвязного списка в обратном порядке.

Сделайте два решения: с использованием цикла и через рекурсию.

К решению

Остаточные параметры и оператор расширения

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов.

Например:

- `Math.max(arg1, arg2, ..., argN)` – вычисляет максимальное число из переданных.
- `Object.assign(dest, src1, ..., srcN)` – копирует свойства из исходных объектов `src1..N` в целевой объект `dest`.
- …и так далее.

В этой главе мы узнаем, как сделать то же самое с нашими собственными функциями и как передавать таким функциям параметры в виде массива.

Остаточные параметры (...)

Вызывать функцию можно с любым количеством аргументов независимо от того, как она была определена.

Например:

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

Лишние аргументы не вызовут ошибку. Но, конечно, посчитаются только первые два.

Остаточные параметры могут быть обозначены через три точки `...`. Буквально это значит: «собери оставшиеся параметры и положи их в массив».

Например, соберём все аргументы в массив `args`:

```
function sumAll(...args) { // args — имя массива
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Мы можем положить первые несколько параметров в переменные, а остальные – собрать в массив.

В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив `titles`:

```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Юлий Цезарь

  // Оставшиеся параметры пойдут в массив
  // titles = ["Консул", "Император"]
  alert( titles[0] ); // Консул
  alert( titles[1] ); // Император
  alert( titles.length ); // 2
}

showName("Юлий", "Цезарь", "Консул", "Император");
```

Остаточные параметры должны располагаться в конце

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них. Это вызовет ошибку:

```
function f(arg1, ...rest, arg2) { // arg2 после
```

```
  ...rest ?!
    // Ошибка
  }
```

`...rest` должен всегда быть последним.

Переменная "arguments"

Все аргументы функции находятся в псевдомассиве `arguments` под своими порядковыми номерами.

Например:

```
function showName() {
  alert( arguments.length );
  alert( arguments[0] );
  alert( arguments[1] );

  // Объект arguments можно перебирать
  // for (let arg of arguments) alert(arg);
}

// Вывод: 2, Юлий, Цезарь
showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined (второго аргумента нет)
showName("Илья");
```

Раньше в языке не было остаточных параметров, и получить все аргументы функции можно было только с помощью `arguments`. Этот способ всё ещё работает, мы можем найти его в старом коде.

Но у него есть один недостаток. Хотя `arguments` похож на массив, и его тоже можно перебирать, это всё же не массив. Он не поддерживает методы массивов, поэтому мы не можем, например, вызвать `arguments.map(...)`.

К тому же, `arguments` всегда содержит все аргументы функции — мы не можем получить их часть. А остаточные параметры позволяют это сделать.

Соответственно, для более удобной работы с аргументами лучше использовать остаточные параметры.

Стрелочные функции не имеют `"arguments"`

Если мы обратимся к `arguments` из стрелочной функции, то получим аргументы внешней «нормальной» функции.

Пример:

```
function f() {
  let showArg = () => alert(arguments[0]);
  showArg(2);
}

f(1); // 1
```

Как мы помним, у стрелочных функций нет собственного `this`. Теперь мы знаем, что нет и своего объекта `arguments`.

Оператор расширения

Мы узнали, как получить массив из списка параметров.

Но иногда нужно сделать в точности противоположное.

Например, есть встроенная функция Math.max. Она возвращает наибольшее число из списка:

```
alert( Math.max(3, 5, 1) ); // 5
```

Допустим, у нас есть массив чисел `[3, 5, 1]`. Как вызвать для него `Math.max`?

Просто так их не вставишь — `Math.max` ожидает получить список чисел, а не один массив.

```
let arr = [3, 5, 1];

alert( Math.max(arr) ); // NaN
```

Конечно, мы можем вводить числа вручную : `Math.max(arr[0], arr[1], arr[2])`. Но, во-первых, это плохо выглядит, а, во-вторых, мы не всегда знаем, сколько будет аргументов. Их может быть как очень много, так и не быть совсем.

И тут нам поможет *оператор расширения*. Он похож на остаточные параметры – тоже использует `...`, но делает совершенно противоположное.

Когда `...arr` используется при вызове функции, он «расширяет» перебираемый объект `arr` в список аргументов.

Для `Math.max`:

```
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (оператор "раскрывает" массив в список аргументов)
```

Этим же способом мы можем передать несколько итерируемых объектов:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8
```

Мы даже можем комбинировать оператор расширения с обычными значениями:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Оператор расширения можно использовать и для слияния массивов:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];



alert(merged); // 0,3,5,1,2,8,9,15 (0, затем arr, затем 2, в конце arr2)
let merged = [0, ...arr, 2, ...arr2];
```

В примерах выше мы использовали массив, чтобы продемонстрировать свойства оператора расширения, но он работает с любым перебираемым объектом.

Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

```
let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т
```

Посмотрим, что происходит. Под капотом оператор расширения использует итераторы, чтобы перебирать элементы. Так же, как это делает `for..of`.

Цикл `for..of` перебирает строку как последовательность символов, поэтому из `...str` получается `"П", "р", "и", "в", "е", "т"`. Получившиеся символы собираются в массив при помощи стандартного объявления массива: `[...str]`.

Для этой задачи мы можем использовать и `Array.from`. Он тоже преобразует перебираемый объект (такой как строка) в массив:

```
let str = "Привет";

// Array.from преобразует перебираемый объект в массив
alert( Array.from(str) ); // П,р,и,в,е,т
```

Результат аналогичен `[...str]`.

Но между `Array.from(obj)` и `[...obj]` есть разница:

- `Array.from` работает как с псевдомассивами, так и с итерируемыми объектами
- Оператор расширения работает только с итерируемыми объектами

Выходит, что если нужно сделать из чего угодно массив, то `Array.from` — более универсальный метод.

Итого

Когда мы видим `"..."` в коде, это могут быть как остаточные параметры, так и оператор расширения.

Как отличить их друг от друга:

- Если `...` располагается в конце списка аргументов функции, то это «остаточные параметры». Он собирает остальные неуказанные аргументы и делает из них массив.
- Если `...` встретился в вызове функции или где-либо ещё, то это «оператор расширения». Он извлекает элементы из массива.

Полезно запомнить:

- Остаточные параметры используются, чтобы создавать новые функции с неопределённым числом аргументов.
- С помощью оператора расширения можно вставить

массив в функцию, которая по умолчанию работает с обычным списком аргументов.

Вместе эти конструкции помогают легко преобразовывать наборы значений в массивы и обратно.

К аргументам функции можно обращаться и по-старому — через псевдомассив `arguments`.

Замыкание

JavaScript – язык с сильным функционально-ориентированным уклоном. Он даёт нам много свободы. Функция может быть динамически создана, скопирована в другую переменную или передана как аргумент другой функции и позже вызвана из совершенно другого места.

Мы знаем, что функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Но что произойдёт, когда внешние переменные изменятся? Функция получит последнее значение или то, которое существовало на момент создания функции?

И что произойдёт, когда функция переместится в другое место в коде и будет вызвана оттуда – получит ли она доступ к внешним переменным своего нового местоположения?

Разные языки ведут себя по-разному в таких случаях, и в этой главе мы рассмотрим поведение JavaScript.

Пара вопросов

Для начала давайте рассмотрим две ситуации, а затем изучим внутренние механизмы шаг за шагом, чтобы вы смогли ответить на эти и более сложные вопросы в будущем.

1. Функция `sayHi` использует внешнюю переменную `name`. Какое значение будет использовать функция при выполнении?

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete";

sayHi(); // что будет показано: "John" или "Pete"?
```

   Такие ситуации распространены и в браузерной и в серверной разработке. Выполнение функции может быть запланировано позже, чем она была создана, например, после какого-нибудь пользовательского действия или сетевого запроса.

   Итак, вопрос в том, получит ли она доступ к последним изменениям?

2. Функция `makeWorker` создаёт другую функцию и возвращает её. Новая функция может быть вызвана откуда-то ещё. Получит ли она доступ к внешним переменным из места своего создания или места выполнения или из обоих?

```
function makeWorker() {
```

```
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";

// create a function
let work = makeWorker();

// call it
work(); // что будет показано? "Pete" (из места создания) или "John" (из места выполнения)
```

Лексическое Окружение

Чтобы понять, что происходит, давайте для начала обсудим, что такое «переменная» на самом деле.

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый *лексическим окружением* `LexicalEnvironment`.

Объект лексического окружения состоит из двух частей:

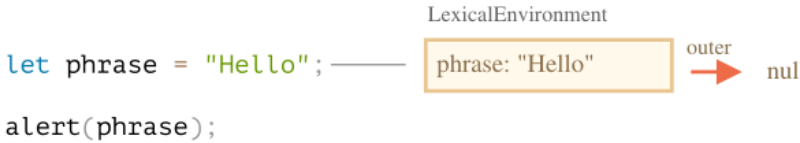
1. *Environment Record* – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение `this`).

2. Ссылка на *внешнее лексическое окружение* – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

"Переменная" – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или

изменить свойство этого объекта».

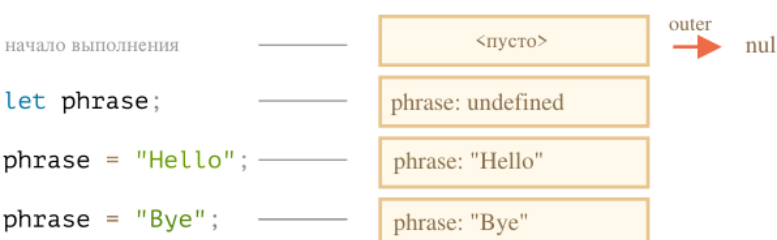
Например, в этом простом коде только одно лексическое окружение:



Это, так называемое, глобальное лексическое окружение, связанное со всем скриптом.

На картинке выше прямоугольник означает Environment Record (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на `null`.

А вот как оно изменяется при объявлении и присваивании переменной:



Прямоугольники с правой стороны демонстрируют, как глобальное лексическое окружение изменяется в процессе выполнения кода:

1. В начале скрипта лексическое окружение пустое.
2. Появляется определение переменной `let phrase`. У неё нет присвоенного значения, поэтому присваивается `undefined`.

3. Переменной `phrase` присваивается значение.
4. Переменная `phrase` меняет значение.

Пока что всё выглядит просто, правда?

Итого:

> Переменная – это свойство специального внутреннего объекта, связанного с текущим выполняющимся блоком/функцией/скриптом.
>
> Работа с переменными – это на самом деле работа со свойствами этого объекта.

Function Declaration

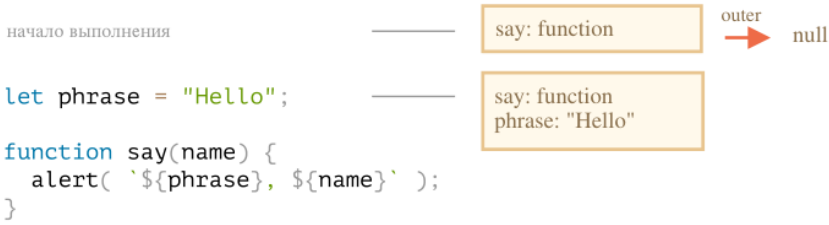
До сих пор мы рассматривали только переменные. Теперь рассмотрим Function Declaration.

В отличие от переменных, объявленных с помощью `let`, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение.

Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение.

Вот почему мы можем вызвать функцию, объявленную через Function Declaration, до того, как она определена.

Следующий код демонстрирует, что уже с самого начала в лексическом окружении что-то есть. Там есть `say`, потому что это Function Declaration. И позже там появится `phrase`, объявленное через `let`:



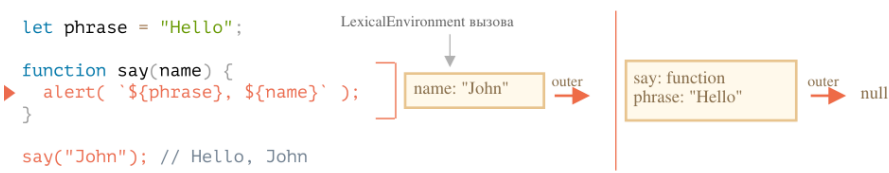
Внутреннее и внешнее лексическое окружение

Теперь давайте продолжим и посмотрим, что происходит, когда функция получает доступ к внешней переменной.

В течение вызова `say()` использует внешнюю переменную `phrase`. Давайте разберёмся подробно, что происходит.

При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова.

Например, для `say("John")` это выглядит так (выполнение находится на строке, отмеченной стрелкой):



Итак, в процессе вызова функции у нас есть два лексических окружения: внутреннее (для вызываемой функции) и внешнее (глобальное):

> Внутреннее лексическое окружение соответствует текущему выполнению `say`.
>
> В нём находится одна переменная `name`, аргумент функции. Мы вызываем `say("John")`, так что значение переменной `name` равно `"John"`.

Внешнее лексическое окружение – это глобальное лексическое окружение.

В нём находятся переменная `phrase` и сама функция.

У внутреннего лексического окружения есть ссылка `outer` на внешнее.

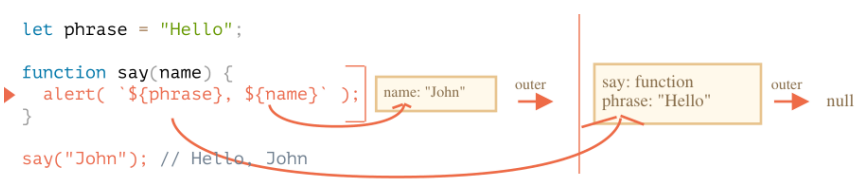
Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.

Если переменная не была найдена, это будет ошибкой в `strict mode`. Без `strict mode`, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким именем.

Давайте посмотрим, как происходит поиск в нашем примере:

Когда `alert` внутри `say` хочет получить доступ к `name`, он немедленно находит переменную в лексическом окружении функции.

Когда он хочет получить доступ к `phrase`, которой нет локально, он следует дальше по ссылке к внешнему лексическому окружению и находит переменную там.



Теперь у нас есть ответ на первый вопрос из начала главы.

Функция получает текущее значение внешних

переменных, то есть, их последнее значение

Старые значения переменных нигде не сохраняются. Когда функция хочет получить доступ к переменной, она берёт её текущее значение из своего или внешнего лексического окружения.

Так что, ответ на первый вопрос: `Pete`:

```
let name = "John";

function sayHi() {
  alert("Hi, " + name);
}

name = "Pete"; // (*)

sayHi(); // Pete
```

Порядок выполнения кода, приведённого выше:

1. В глобальном лексическом окружении есть `name: "John"`.
2. На строке `(*)` глобальная переменная изменяется, теперь `name: "Pete"`.
3. Момент, когда выполняется функция `sayHi()` и берёт переменную `name` извне. Теперь из глобального лексического окружения, где переменная уже равна `"Pete"`.

Один вызов – одно лексическое окружение

Пожалуйста, обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется.

И, если функция вызывается несколько раз, то

для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

Лексическое окружение – это специальный внутренний объект

«Лексическое окружение» – это специальный внутренний объект. Мы не можем получить его в нашем коде и изменять напрямую. Сам движок JavaScript может оптимизировать его, уничтожать неиспользуемые переменные для освобождения памяти и выполнять другие внутренние уловки, но видимое поведение объекта должно оставаться таким, как было описано.

Вложенные функции

Функция называется «вложенной», когда она создаётся внутри другой функции.

Это очень легко сделать в JavaScript.

Мы можем использовать это для упорядочивания нашего кода, например, как здесь:

```
function sayHiBye(firstName, lastName) {

  // функция-помощник, которую мы используем ниже
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );
```

```
}
```

Здесь *вложенная* функция `getFullName()` создана для удобства. Она может получить доступ к внешним переменным и, значит, вывести полное имя. В JavaScript вложенные функции используются очень часто.

Что ещё интереснее, вложенная функция может быть возвращена: либо в качестве свойства нового объекта (если внешняя функция создаёт объект с методами), либо сама по себе. И затем может быть использована в любом месте. Не важно где, она всё так же будет иметь доступ к тем же внешним переменным.

Например, здесь, вложенная функция присваивается новому объекту в конструкторе:

```
// функция-конструктор возвращает новый объект
function User(name) {

  // методом объекта становится вложенная функция
  this.sayHi = function() {
    alert(name);
  };
}

let user = new User("John");
user.sayHi(); // у кода метода "sayHi" есть доступ к внешней
переменной "name"
```

А здесь мы просто создаём и возвращаем функцию «счётчик»:

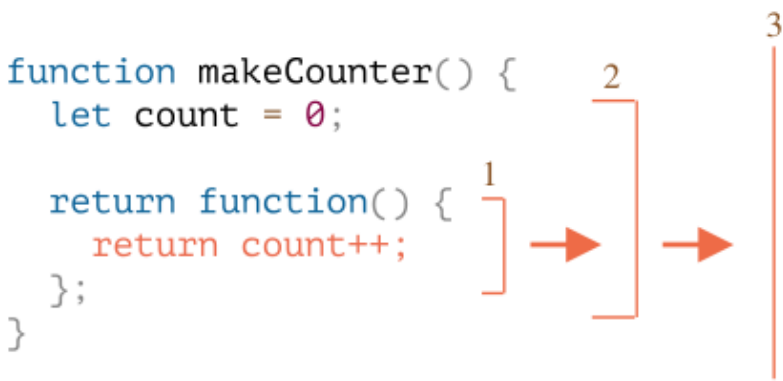
```
function makeCounter() {
  let count = 0;

  return function() {
    return count++; // есть доступ к внешней переменной
"count"
  };
}

let counter = makeCounter();
```

```
alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Давайте продолжим с примером `makeCounter`. Он создаёт функцию «counter», которая возвращает следующее число при каждом вызове. Несмотря на простоту, немного модифицированные варианты этого кода применяются на практике, например, в генераторе псевдослучайных чисел и во многих других случаях.

Как же это работает изнутри?

Когда внутренняя функция начинает выполняться, начинается поиск переменной `count++` изнутри-наружу. Для примера выше порядок будет такой:



1. Локальные переменные вложенной функции…
2. Переменные внешней функции…
3. И так далее, пока не будут достигнуты глобальные переменные.

В этом примере `count` будет найден на шаге `2`. Когда внешняя переменная модифицируется, она изменится там, где была найдена. Значит, `count++` найдёт внешнюю переменную и увеличит её значение в лексическом окружении, которому она принадлежит. Как если бы у нас было `let count = 1`.

Теперь рассмотрим два вопроса:

1. Можем ли мы каким-нибудь образом сбросить счётчик `count` из кода, который не принадлежит `makeCounter`? Например, после вызова `alert` в коде выше.
2. Если мы вызываем `makeCounter` несколько раз – нам возвращается много функций `counter`. Они независимы или разделяют одну и ту же переменную `count`?

Попробуйте ответить на эти вопросы перед тем, как продолжить чтение.

…

Готовы?

Хорошо, давайте ответим на вопросы.

1. Такой возможности нет: `count` – локальная переменная функции, мы не можем получить к ней доступ извне.
2. Для каждого вызова `makeCounter()` создаётся новое лексическое окружение функции, со своим собственным `count`. Так что, получившиеся функции `counter` – независимы.

Вот демо:

```
function makeCounter() {
  let count = 0;
  return function() {
    return count++;
  };
}

let counter1 = makeCounter();
let counter2 = makeCounter();

alert( counter1() ); // 0
```

```
alert( counter1() ); // 1

alert( counter2() ); // 0 (независимо)
```

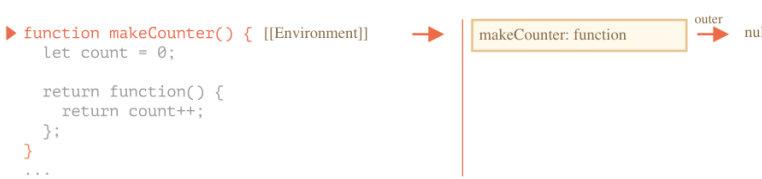
Надеюсь, ситуация с внешними переменными теперь ясна. Для большинства ситуаций такого понимания вполне достаточно, но в спецификации есть ряд деталей, которые мы, для простоты, опустили. Далее мы разберём происходящее ещё более подробно.

Окружение в деталях

Вот что происходит в примере с `makeCounter` шаг за шагом. Пройдите их, чтобы убедиться, что вы разобрались с каждой деталью.

Пожалуйста, обратите внимание на дополнительное свойство `[[Environment]]`, про которое здесь рассказано. Мы не упоминали о нём раньше для простоты.

1. Когда скрипт только начинает выполняться, есть только глобальное лексическое окружение:



   В этот начальный момент есть только функция `makeCounter`, потому что это Function Declaration. Она ещё не выполняется.

   Все функции «при рождении» получают скрытое свойство `[[Environment]]`, которое ссылается на лексическое окружение места, где они были созданы.

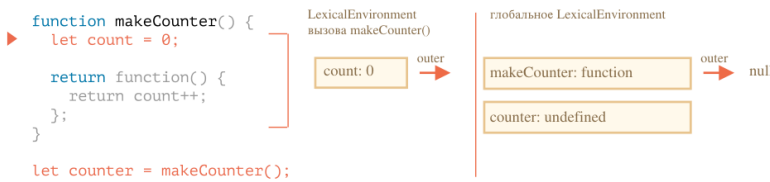
   Мы ещё не говорили об этом, это то, каким образом

функции знают, где они были созданы.

В данном случае, `makeCounter` создан в глобальном лексическом окружении, так что `[[Environment]]` содержит ссылку на него.

Другими словами, функция навсегда запоминает ссылку на лексическое окружение, где она была создана. И `[[Environment]]` – скрытое свойство функции, которое содержит эту ссылку.

2. Код продолжает выполняться, объявляется новая глобальная переменная `counter`, которой присваивается результат вызова `makeCounter`. Вот снимок момента, когда интерпретатор находится на первой строке внутри `makeCounter()`:



В момент вызова `makeCounter()` создаётся лексическое окружение, для хранения его переменных и аргументов.

Как и все лексические окружения, оно содержит две вещи:

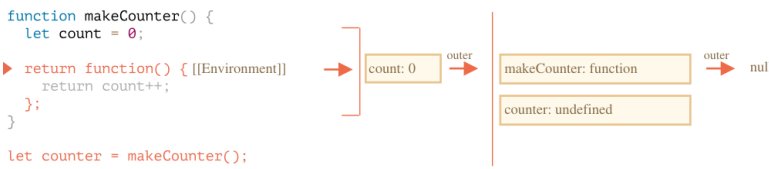
1. Environment Record с локальными переменными. В нашем случае `count` – единственная локальная переменная (появляющаяся, когда выполняется строчка с `let count`).
2. Ссылка на внешнее окружение, которая устанавливается в значение `[[Environment]]` функции. В данном случае, `[[Environment]]` функции `makeCounter` ссылается на глобальное лексическое окружение.

Итак, теперь у нас есть два лексических окружения: первое – глобальное, второе – для текущего вызова `makeCounter`, с внешней ссылкой на глобальный объект.

3. В процессе выполнения `makeCounter()` создаётся небольшая вложенная функция.

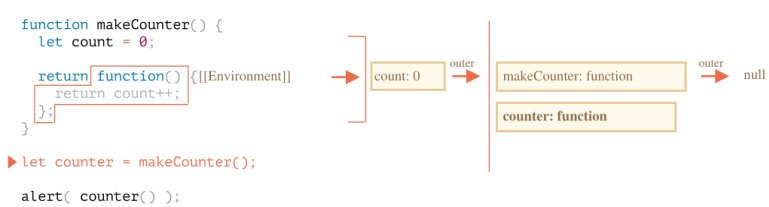
   Не имеет значения, какой способ объявления функции используется: Function Declaration или Function Expression. Все функции получают свойство `[[Environment]]`, которое ссылается на лексическое окружение, в котором они были созданы. То же самое происходит и с нашей новой маленькой функцией.

   Для нашей новой вложенной функции значением `[[Environment]]` будет текущее лексическое окружение `makeCounter()` (где она была создана):



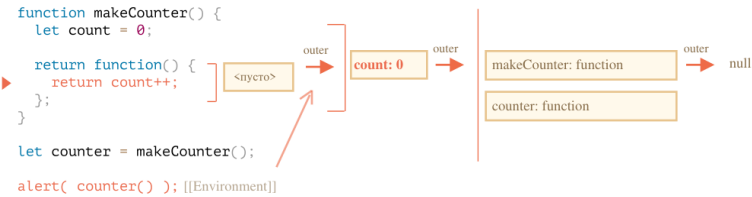
   Пожалуйста, обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри `function() { return count++ }` не выполняется.

4. Выполнение продолжается, вызов `makeCounter()` завершается, и результат (небольшая вложенная функция) присваивается глобальной переменной `counter`:



В этой функции есть только одна строчка: `return count++`, которая будет выполнена, когда мы вызовем функцию.

5. При вызове `counter()` для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом `counter` локальных переменных нет. Но `[[Environment]]` `counter` используется, как ссылка на внешнее лексическое окружение `outer`, которое даёт доступ к переменным предшествующего вызова `makeCounter`, где `counter` был создан.



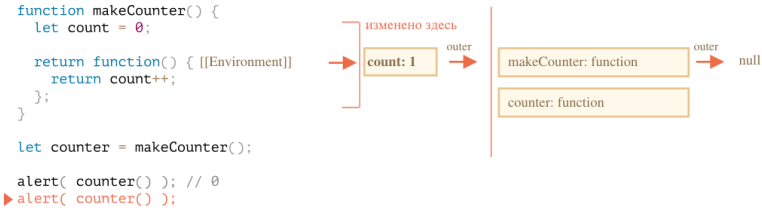
Теперь, когда вызов ищет переменную `count`, он сначала ищет в собственном лексическом окружении (пустое), а затем в лексическом окружении предшествующего вызова `makeCounter()`, где и находит её.

Пожалуйста, обратите внимание, как здесь работает управление памятью. Хотя `makeCounter()` закончил выполнение некоторое время назад, его лексическое окружение остаётся в памяти, потому что есть вложенная функция с `[[Environment]]`, который ссылается на него.

В большинстве случаев, объект лексического окружения существует до того момента, пока есть функция, которая может его использовать. И только тогда, когда таких не остаётся, окружение уничтожается.

6. Вызов `counter()` не только возвращает значение `count`, но также увеличивает его. Обратите внимание,

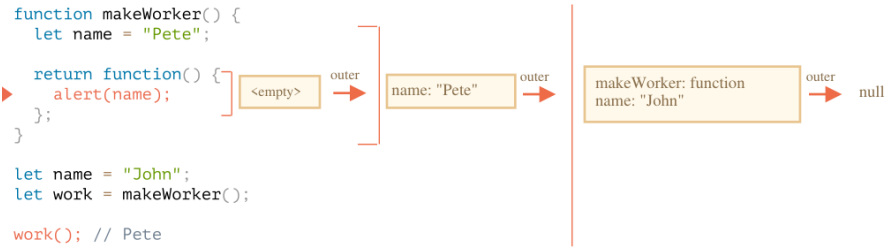
что модификация происходит «на месте». Значение `count` изменяется конкретно в том окружении, где оно было найдено.



7. Следующие вызовы `counter()` сделают то же самое.

Теперь ответ на второй вопрос из начала главы должен быть очевиден.

Функция `work()` в коде ниже получает `name` из того места, где была создана, через ссылку на внешнее лексическое окружение:



Так что, результатом будет `"Pete"`.

Но, если бы в `makeWorker()` не было `let name`, тогда бы поиск продолжился дальше и была бы взята глобальная переменная, как мы видим из приведённой выше цепочки. В таком случае, результатом было бы `"John"`.

Замыкания

В программировании есть общий термин: «замыкание», – которое должен знать каждый разработчик.

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В некоторых языках это невозможно, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но, как было описано выше, в JavaScript, все функции изначально являются замыканиями (есть только одно исключение, про которое будет рассказано в Синтаксис "new Function").

То есть, они автоматически запоминают, где были созданы, с помощью скрытого свойства `[[Environment]]` и все они могут получить доступ к внешним переменным.

Когда на собеседовании фронтенд-разработчик получает вопрос: «что такое замыкание?», – правильным ответом будет определение замыкания и объяснения того факта, что все функции в JavaScript являются замыканиями, и, может быть, несколько слов о технических деталях: свойстве `[[Environment]]` и о том, как работает лексическое окружение.

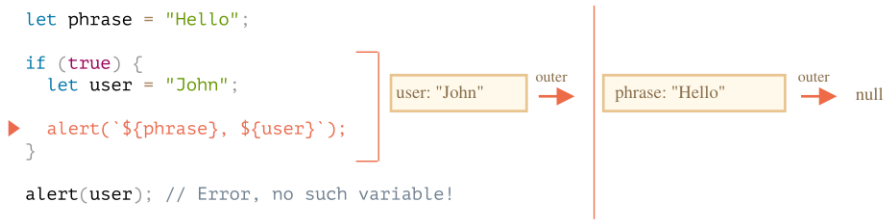
Блоки кода и циклы, IIFE

Предыдущие примеры сосредоточены на функциях. Но лексическое окружение существует для любых блоков кода `{...}`.

Лексическое окружение создаётся при выполнении блока кода и содержит локальные переменные для этого блока. Вот пара примеров.

If

В следующем примере переменная `user` существует только в блоке `if`:



Когда выполнение попадает в блок `if`, для этого блока создаётся новое лексическое окружение.

У него есть ссылка на внешнее окружение, так что `phrase` может быть найдена. Но все переменные и Function Expression, объявленные внутри `if`, остаются в его лексическом окружении и не видны снаружи.

Например, после завершения `if` следующий `alert` не увидит `user`, что вызовет ошибку.

For, while

Для цикла у каждой итерации своё отдельное лексическое окружение. Если переменная объявлена в `for(let ...)`, то она также в нём:

```
for (let i = 0; i < 10; i++) {
  // У каждой итерации цикла своё собственное лексическое окружение
  // {i: value}
}

alert(i); // Ошибка, нет такой переменной
```

Обратите внимание: `let i` визуально находится снаружи `{...}`. Но конструкция `for` – особенная в этом смысле, у каждой итерации цикла своё собственное лексическое окружение с текущим `i` в нём.

И так же, как и в `if`, ниже цикла `i` невидима.

Блоки кода

Мы также можем использовать «простые» блоки кода `{...}`, чтобы изолировать переменные в «локальной области видимости».

Например, в браузере все скрипты (кроме `type="module"`) разделяют одну общую глобальную область. Так что, если мы создадим глобальную переменную в одном скрипте, она станет доступна и в других. Но это становится источником конфликтов, если два скрипта используют одно и то же имя переменной и перезаписывают друг друга.

Это может произойти, если название переменной – широко распространённое слово, а авторы скрипта не знают друг о друге.

Если мы хотим этого избежать, мы можем использовать блок кода для изоляции всего скрипта или какой-то его части:

```
{
  // сделать какую-нибудь работу с локальными переменными, которые не должны быть видны снаружи

  let message = "Hello";

  alert(message); // Hello
}

alert(message); // Ошибка: переменная message не определена
```

Из-за того, что у блока есть собственное лексическое окружение, код снаружи него (или в другом скрипте) не видит переменные этого блока.

IIFE

В прошлом в JavaScript не было лексического окружения на уровне блоков кода.

Так что программистам пришлось что-то придумать. И то, что они сделали, называется «immediately-invoked function expressions» (аббревиатура IIFE), что означает функцию, запускаемую сразу после объявления.

Это не то, что мы должны использовать сегодня, но, так как вы можете встретить это в старых скриптах, полезно понимать принцип работы.

IIFE выглядит так:

```
(function() {

  let message = "Hello";

  alert(message); // Hello

})();
```

Здесь создаётся и немедленно вызывается Function Expression. Так что код выполняется сразу же и у него есть свои локальные переменные.

Function Expression обёрнуто в скобки `(function {...})`, потому что, когда JavaScript встречает `"function"` в основном потоке кода, он воспринимает это как начало Function Declaration. Но у Function Declaration должно быть имя, так что такой код вызовет ошибку:

```
// Попробуйте объявить и сразу же вызвать функцию
function() { // <-- Error: Unexpected token (

  let message = "Hello";

  alert(message); // Hello

}();
```

Даже если мы скажем: «хорошо, давайте добавим имя», – это не сработает, потому что JavaScript не позволяет вызывать Function Declaration немедленно.

```
// ошибка синтаксиса из-за скобок ниже
function go() {

}(); // <-- не можете вызывать Function Declaration немедленно
```

Так что, скобки вокруг функции – это трюк, который позволяет показать JavaScript, что функция была создана в контексте другого выражения, и, таким образом, это функциональное выражение: ей не нужно имя и её можно вызвать немедленно.

Кроме скобок, существуют и другие пути показать JavaScript, что мы имеем в виду Function Expression:

```
// Пути создания IIFE

(function() {
  alert("Скобки вокруг функции");
}());

(function() {
  alert("Скобки вокруг всего");
}());

function() {
  alert("Выражение начинается с логического оператора NOT");
}();

function() {
  alert("Выражение начинается с унарного плюса");
}();
)
)
!
+
```

Во всех перечисленных случаях мы объявляем Function Expression и немедленно выполняем его. Ещё раз заметим, что в настоящий момент нет необходимости писать подобный код.

Сборка мусора

Обычно лексическое окружение очищается и удаляется после того, как функция выполнилась. Например:

```
function f() {
  let value1 = 123;
  let value2 = 456;
}

f();
```

Здесь два значения, которые технически являются свойствами лексического окружения. Но после того, как `f()` завершится, это лексическое окружение станет недоступно, поэтому оно удалится из памяти.

…Но, если есть вложенная функция, которая всё ещё доступна после выполнения `f`, то у неё есть свойство `[[Environment]]`, которое ссылается на внешнее лексическое окружение, тем самым оставляя его достижимым, «живым»:

```
function f() {
  let value = 123;

  function g() { alert(value); }


}

let g = f(); // g доступно и продолжает держать внешнее
лексическое окружение в памяти
   return g;
```

Обратите внимание, если `f()` вызывается несколько раз и возвращаемые функции сохраняются, тогда все соответствующие объекты лексического окружения продолжат держаться в памяти. Вот три такие функции в коде ниже:

```
function f() {
  let value = Math.random();
```

```
  return function() { alert(value); };
}

// три функции в массиве, каждая из них ссылается на
лексическое окружение
// из соответствующего вызова f()
let arr = [f(), f(), f()];
```

Объект лексического окружения умирает, когда становится недоступным (как и любой другой объект). Другими словами, он существует только до того момента, пока есть хотя бы одна вложенная функция, которая ссылается на него.

В следующем коде, после того как `g` станет недоступным, лексическое окружение функции (и, соответственно, `value`) будет удалено из памяти;

```
function f() {
  let value = 123;

  function g() { alert(value); }

  return g;
}

let g = f(); // пока g существует,
// соответствующее лексическое окружение существует

g = null; // ...а теперь память очищается
```

Оптимизация на практике

Как мы видели, в теории, пока функция жива, все внешние переменные тоже сохраняются.

Но на практике движки JavaScript пытаются это оптимизировать. Они анализируют использование переменных и, если легко по коду понять, что внешняя переменная не используется – она удаляется.

Одним из важных побочных эффектов в V8 (Chrome, Opera) является то, что такая переменная становится недоступной при отладке.

Попробуйте запустить следующий пример в Chrome с открытой Developer Tools.

Когда код будет поставлен на паузу, напишите в консоли `alert(value)`.

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // в консоли: напишите alert(value); Такой
переменной нет!
  }

  return g;
}

let g = f();
g();
```

Как вы можете видеть – такой переменной не существует! В теории, она должна быть доступна, но попала под оптимизацию движка.

Это может приводить к забавным (если удаётся решить быстро) проблемам при отладке. Одна из них – мы можем увидеть не ту внешнюю переменную при совпадающих названиях:

```
let value = "Сюрприз!";

function f() {
  let value = "ближайшее значение";

  function g() {
    debugger; // в консоли: напишите alert(value); Сюрприз!
  }

  return g;
}

let g = f();
g();
```

До встречи!

Эту особенность V8 полезно знать. Если вы занимаетесь отладкой в Chrome/Opera, рано или поздно вы с ней встретитесь.

Это не баг в отладчике, а скорее особенность V8. Возможно со временем это изменится. Вы всегда можете проверить это, запустив пример на этой странице.

Задачи

Независимы ли счётчики?

важность: 5

Здесь мы делаем два счётчика: `counter` и `counter2`, используя одну и ту же функцию `makeCounter`.

Они независимы? Что покажет второй счётчик? `0,1` или `2,3` или что-то ещё?

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
let counter2 = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1

alert( counter2() ); // ?
alert( counter2() ); // ?
```

К решению

Объект счётчика

важность: 5

Здесь объект счётчика создан с помощью функции-конструктора.

Будет ли он работать? Что покажет?

```
function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };
  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // ?
alert( counter.up() ); // ?
alert( counter.down() ); // ?
```

К решению

Функция в if

Посмотрите на код. Какой будет результат у вызова на последней строке?

```
let phrase = "Hello";

if (true) {
  let user = "John";

  function sayHi() {
    alert(`${phrase}, ${user}`);
```

```
  }
}

sayHi();
```

К решению

Сумма с помощью замыканий

важность: 4

Напишите функцию `sum`, которая работает таким образом: `sum(a)(b) = a+b`.

Да, именно таким образом, используя двойные круглые скобки (не опечатка).

Например:

```
sum(1)(2) = 3
sum(5)(-1) = 4
```

К решению

Фильтрация с помощью функции

важность: 5

У нас есть встроенный метод `arr.filter(f)` для массивов. Он фильтрует все элементы с помощью функции `f`. Если она возвращает `true`, то элемент добавится в возвращаемый массив.

Сделайте набор «готовых к употреблению» фильтров:

`inBetween(a, b)` – между `a` и `b` (включительно).

`inArray([...])` – находится в данном массиве.

Они должны использоваться таким образом:

`arr.filter(inBetween(3,6))` – выбирает только значения между 3 и 6 (включительно).

`arr.filter(inArray([1,2,3]))` – выбирает только элементы, совпадающие с одним из элементов массива

Например:

```
/* .. ваш код для inBetween и inArray */
let arr = [1, 2, 3, 4, 5, 6, 7];

alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6

alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Открыть песочницу с тестами для задачи.

К решению

Сортировать по полю

важность: 5

У нас есть массив объектов, который нужно отсортировать:

```
let users = [
  { name: "John", age: 20, surname: "Johnson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];
```

Обычный способ был бы таким:

```
// по имени (Ann, John, Pete)
```

```
users.sort((a, b) => a.name > b.name ? 1 : -1);

// по возрасту (Pete, Ann, John)
users.sort((a, b) => a.age > b.age ? 1 : -1);
```

Можем ли мы сделать его короче, скажем, вот таким?

```
users.sort(byField('name'));
users.sort(byField('age'));
```

То есть, чтобы вместо функции, мы просто писали `byField(fieldName)`.

Напишите функцию `byField`, которая может быть использована для этого.

К решению

Армия функций

важность: 5

Следующий код создаёт массив из стрелков (`shooters`).

Каждая функция предназначена выводить их порядковые номера. Но что-то пошло не так…

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {
    let shooter = function() { // функция shooter
      alert( i ); // должна выводить порядковый номер
    };
    shooters.push(shooter);
    i++;
  }

  return shooters;
}
```

```
let army = makeArmy();

army[0](); // у 0-го стрелка будет номер 10
army[5](); // и у 5-го стрелка тоже будет номер 10
// ... у всех стрелков будет номер 10, вместо 0, 1, 2, 3...
```

Почему у всех стрелков одинаковые номера? Почините код, чтобы он работал как задумано.

Открыть песочницу с тестами для задачи.

К решению

Устаревшее ключевое слово "var"

В самой первой главе про переменные мы ознакомились с тремя способами объявления переменных:

1. `let`
2. `const`
3. `var`

`let` и `const` ведут себя одинаково по отношению к лексическому окружению, области видимости.

Но `var` – это совершенно другой зверь, берущий своё начало с давних времён. Обычно `var` не используется в современных скриптах, но всё ещё может скрываться в старых.

Если в данный момент вы не работаете с подобными скриптами, вы можете пропустить или отложить прочтение данной главы, однако, есть шанс, что вы столкнётесь с

`var` в будущем.

На первый взгляд, поведение `var` похоже на `let`. Например, объявление переменной:

```
function sayHi() {
  var phrase = "Привет"; // локальная переменная, "var" вместо
"let"

  alert(phrase); // Привет
}

sayHi();

alert(phrase); // Ошибка: phrase не определена
```

…Однако, отличия всё же есть.

Для «var» не существует блочной области видимости

Область видимости переменных `var` ограничивается либо функцией, либо, если переменная глобальная, то скриптом. Такие переменные доступны за пределами блока.

Например:

```
if (true) {
  var test = true; // используем var вместо let
}

alert(test); // true, переменная существует
вне блока if
```

Так как `var` игнорирует блоки, мы получили глобальную переменную `test`.

А если бы мы использовали `let test` вместо `var test`, тогда переменная была бы видна только внутри `if`:

```
if (true) {
  let test = true; // используем let
}

alert(test); // Error: test is not defined
```

Аналогично для циклов: `var` не может быть блочной или локальной внутри цикла:

```
for (var i = 0; i < 10; i++) {
  // ...
}

alert(i); // 10, переменная i доступна вне цикла, т.к. является глобальной переменной
```

Если блок кода находится внутри функции, то `var` становится локальной переменной в этой функции:

```
function sayHi() {
  if (true) {
    var phrase = "Привет";
  }

  alert(phrase); // срабатывает и выводит "Привет"
}

sayHi();
alert(phrase); // Ошибка: phrase не определена (видна в консоли разработчика)
```

Как мы видим, `var` выходит за пределы блоков `if`, `for` и подобных. Это происходит потому, что на заре развития JavaScript блоки кода не имели лексического окружения. Поэтому можно сказать, что `var` – это пережиток прошлого.

«var» обрабатываются в начале запуска функции

Объявления переменных `var` обрабатываются в начале выполнения функции (или запуска скрипта, если переменная является глобальной).

Другими словами, переменные `var` считаются объявленными с самого начала исполнения функции вне зависимости от того, в каком месте функции реально находятся их объявления (при условии, что они не находятся во вложенной функции).

Т.е. этот код:

```
function sayHi() {
  phrase = "Привет";

  alert(phrase);


}
sayHi();
  var phrase;
```

…Технически полностью эквивалентен следующему (объявление переменной `var phrase` перемещено в начало функции):

```
function sayHi() {


  phrase = "Привет";

  alert(phrase);
}
sayHi();
  var phrase;
```

…И даже коду ниже (как вы помните, блочная область видимости игнорируется):

```
function sayHi() {
  phrase = "Привет"; // (*)
```

```
  alert(phrase);
}
sayHi();
  if (false) {
    var phrase;
  }
```

Это поведение называется «hoisting» (всплытие, поднятие), потому что все объявления переменных `var` «всплывают» в самый верх функции.

В примере выше `if (false)` условие никогда не выполнится. Но это никаким образом не препятствует созданию переменной `var phrase`, которая находится внутри него, поскольку объявления `var` «всплывают» в начало функции. Т.е. в момент присвоения значения `(*)` переменная уже существует.

Объявления переменных «всплывают», но присваивания значений – нет.

Это проще всего продемонстрировать на примере:

```
function sayHi() {
  alert(phrase);


}

sayHi();
  var phrase = "Привет";
```

Строка `var phrase = "Привет"` состоит из двух действий:

1. Объявление переменной `var`
2. Присвоение значения в переменную `=`.

Объявление переменной обрабатывается в начале

выполнения функции («всплывает»), однако присвоение значения всегда происходит в той строке кода, где оно указано. Т.е. код выполняется по следующему сценарию:

```
function sayHi() {

  alert(phrase); // undefined

}

sayHi();
  var phrase; // объявление переменной срабатывает вначале...
  phrase = "Привет"; // ...присвоение - в момент, когда исполнится данная строка кода.
```

Поскольку все объявления переменных `var` обрабатываются в начале функции, мы можем ссылаться на них в любом месте. Однако, переменные имеют значение `undefined` до строки с присвоением значения.

В обоих примерах выше вызов `alert` происходил без ошибки, потому что переменная `phrase` уже существовала. Но её значение ещё не было присвоено, поэтому мы получали `undefined`.

Итого

Существует 2 основных отличия `var` от `let/const`:

1. Переменные `var` не имеют блочной области видимости, они ограничены, как минимум, телом функции.
2. Объявления (инициализация) переменных `var`производится в начале исполнения функции (или скрипта для глобальных переменных).

Есть ещё одно небольшое отличие, относящееся к глобальному объекту, мы рассмотрим его в следующей главе.

Эти особенности, как правило, не очень хорошо влияют на код. Блочная область видимости – это удобно. Поэтому много лет назад `let` и `const` были введены в стандарт и сейчас являются основным способом объявления переменных.

Глобальный объект

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.

В браузере он называется `window`, в Node.js — `global`, в другой среде исполнения может называться иначе.

Недавно `globalThis` был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. В некоторых браузерах, например Edge не на Chromium, `globalThis` ещё не поддерживается, но легко реализуется с помощью полифила.

Далее мы будем использовать `window`, полагая, что наша среда – браузер. Если скрипт может выполняться и в другом окружении, лучше будет `globalThis`.

Ко всем свойствам глобального объекта можно обращаться напрямую:

```
alert("Привет");
// это то же самое, что и
window.alert("Привет");
```

В браузере глобальные функции и переменные, объявленные с помощью `var` (не `let/const`!), становятся свойствами глобального объекта:

```
var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)
```

Пожалуйста, не полагайтесь на это. Такое поведение поддерживается для совместимости. В современных проектах, использующих JavaScript-модули, такого не происходит.

Если бы мы объявили переменную при помощи `let`, то такого бы не произошло:

```
let gLet = 5;

alert(window.gLet); // undefined (не становится свойством глобального объекта)
```

Если свойство настолько важное, что вы хотите сделать его доступным для всей программы, запишите его в глобальный объект напрямую:

```
// где угодно в коде
alert(currentUser.name); // John

// или, если у нас есть локальная переменная с именем "currentUser",
// получим её из window явно (безопасно!)
alert(window.currentUser.name); // John
// сделать информацию о текущем пользователе глобальной, для предоставления доступа всем скриптам
window.currentUser = {
  name: "John"
};
```

При этом обычно не рекомендуется использовать глобальные переменные. Следует применять их как можно реже. Дизайн кода, при котором функция получает входные параметры и выдаёт определённый результат, чище, надёжнее и удобнее для тестирования, чем когда используются внешние, а тем более глобальные переменные.

Использование для полифилов

Глобальный объект можно использовать, чтобы проверить поддержку современных возможностей языка.

Например, проверить наличие встроенного объекта `Promise` (такая поддержка отсутствует в очень старых браузерах):

```
if (!window.Promise) {
  alert("Ваш браузер очень старый!");
}
```

Если нет (скажем, используется старый браузер), мы можем создать полифил: добавить функции, которые не поддерживаются окружением, но существуют в современном стандарте.

```
if (!window.Promise) {
  window.Promise = ... // собственная реализация современной возможности языка
}
```

Итого

Глобальный объект хранит переменные, которые должны быть доступны в любом месте программы.

Это включает в себя как встроенные объекты, например, `Array`, так и характерные для окружения свойства, например, `window.innerHeight` – высота окна браузера.

Глобальный объект имеет универсальное имя – `globalThis`.

…Но чаще на него ссылаются по-старому, используя имя, характерное для данного окружения, такое как `window` (браузер) и `global` (Node.js). Так как `globalThis` появился недавно, он не поддерживается в IE и Edge (не-Chromium версия), но можно использовать полифил.

Следует хранить значения в глобальном объекте, только если они действительно глобальны для нашего проекта. И стараться свести их количество к минимуму.

В браузерах, если только мы не используем модули, глобальные функции и переменные, объявленные с помощью `var`, становятся свойствами глобального объекта.

Для того, чтобы код был проще и в будущем его легче было поддерживать, следует обращаться к свойствам глобального объекта напрямую, как `window.x`.

Объект функции, NFE

Как мы уже знаем, в JavaScript функция – это значение.

Каждое значение в JavaScript имеет свой тип. А функция – это какой тип?

В JavaScript функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но и использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

Свойство «name»

Объект функции содержит несколько полезных свойств.

Например, имя функции нам доступно как свойство «name»:

```
function sayHi() {
  alert("Hi");
}

alert(sayHi.name); // sayHi
```

Что довольно забавно, логика назначения `name` весьма умная. Она присваивает корректное имя даже в случае, когда функция создаётся без имени и тут же присваивается, вот так:

```
let sayHi = function() {
  alert("Hi");
};

alert(sayHi.name); // sayHi (есть имя!)
```

Это работает даже в случае присваивания значения по умолчанию:

```
function f(sayHi = function() {}) {
```

```
  alert(sayHi.name); // sayHi (работает!)
}

f();
```

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается определить его из контекста.

Также имена имеют и методы объекта:

```
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }

}

alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

В этом нет никакой магии. Бывает, что корректное имя определить невозможно. В таких случаях свойство name имеет пустое значение. Например:

```
// функция объявлена внутри массива
let arr = [function() {}];

alert( arr[0].name ); // <пустая строка>
// здесь отсутствует возможность определить имя, поэтому его
нет
```

Впрочем, на практике такое бывает редко, обычно функции имеют `name`.

Свойство «length»

Ещё одно встроенное свойство «length» содержит количество параметров функции в её объявлении. Например:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Как мы видим, троеточие, обозначающее «остаточные параметры», здесь как бы «не считается»

Свойство `length` иногда используется для интроспекций в функциях, которые работают с другими функциями.

Например, в коде ниже функция `ask` принимает в качестве параметров вопрос `question` и произвольное количество функций-обработчиков ответа `handler`.

Когда пользователь отвечает на вопрос, функция вызывает обработчики. Мы можем передать два типа обработчиков:

- Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
- Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Чтобы вызвать обработчик `handler` правильно, будем проверять свойство `handler.length`.

Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

```
function ask(question, ...handlers) {
```

```
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

}

// для положительных ответов вызываются оба типа обработчиков
// для отрицательных - только второго типа
ask("Вопрос?", () => alert('Вы ответили да'), result =>
alert(result));
```

Это частный случай так называемого Ad-hoc-полиморфизма – обработка аргументов в зависимости от их типа или, как в нашем случае – от значения `length`. Эта идея имеет применение в библиотеках JavaScript.

Пользовательские свойства

Мы также можем добавить свои собственные свойства.

Давайте добавим свойство `counter` для отслеживания общего количества вызовов:

```
function sayHi() {
  alert("Hi");


}
sayHi.counter = 0; // начальное значение

sayHi(); // Hi
sayHi(); // Hi

alert( `Вызвана ${sayHi.counter} раза` ); // Вызвана 2 раза
  // давайте посчитаем, сколько вызовов мы
сделали
  sayHi.counter++;
```

Свойство не есть переменная

Свойство функции, назначенное как `sayHi.counter = 0`, не *объявляет* локальную переменную `counter` внутри неё. Другими словами, свойство `counter` и переменная `let counter` – это две независимые вещи.

Мы можем использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот. Это два параллельных мира.

Иногда свойства функции могут использоваться вместо замыканий. Например, мы можем переписать функцию-счётчик из главы Замыкание, используя её свойство:

```
function makeCounter() {
  // вместо
  // let count = 0

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

Свойство `count` теперь хранится прямо в функции, а не в её внешнем лексическом окружении.

Это хуже или лучше, чем использовать замыкание?

Основное отличие в том, что если значение `count` живёт во внешней переменной, то оно не доступно для внешнего кода. Изменить его могут только вложенные функции. А если оно присвоено как свойство функции, то мы можем его получить:

```
function makeCounter() {

  function counter() {
    return counter.count++;
  };

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

Поэтому выбор реализации зависит от наших целей.

Named Function Expression

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя.

Например, давайте объявим Function Expression:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
```

И присвоим ему имя:

```
let sayHi = function (who) {
  alert(`Hello, ${who}`);
};
```

`func`

Чего мы здесь достигли? Какова цель этого дополнительного имени `func`?

Для начала заметим, что функция всё ещё задана как Function Expression. Добавление `"func"` после `function` не превращает объявление в Function Declaration, потому что оно все ещё является частью выражения присваивания.

Добавление такого имени ничего не ломает.

Функция все ещё доступна как `sayHi()`:

```
let sayHi = function (who) {
  alert(`Hello, ${who}`);
};

sayHi("John"); // Hello, John
```

`func`

Есть две важные особенности имени `func`, ради которого оно даётся:

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

Например, ниже функция `sayHi` вызывает себя с `"Guest"`, если не передан параметр `who`:

```
let sayHi = function (who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {

  }
};

sayHi(); // Hello, Guest
```

```
// А вот так - не сработает:
func(); // Ошибка, func не определена (недоступна вне функции)
```

`func`

```
    func("Guest"); // использует func, чтобы снова вызвать себя же
```

Почему мы используем `func`? Почему просто не использовать `sayHi` для вложенного вызова?

Вообще, обычно мы можем так поступить:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {

  }
};
```

```
    sayHi("Guest");
```

Однако, у этого кода есть проблема, которая заключается в том, что значение `sayHi` может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнёт выдавать ошибки:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {

  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Ошибка, вложенный вызов sayHi больше не работает!
```

```
    sayHi("Guest"); // Ошибка: sayHi не является функцией
```

Так происходит, потому что функция берёт `sayHi` из внешнего лексического окружения. Так как локальная

переменная `sayHi` отсутствует, используется внешняя. И на момент вызова эта внешняя `sayHi` равна `null`.

Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы.

Давайте используем его, чтобы исправить наш код:

```
let sayHi = function (who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {

  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (вложенный вызов работает)
func
    func("Guest"); // Теперь всё в порядке
```

Теперь всё работает, потому что имя `"func"` локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию.

Внешний код все ещё содержит переменные `sayHi` и `welcome`, но теперь `func` – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Это не работает с Function Declaration

Трюк с «внутренним» именем, описанный выше, работает только для Function Expression и *не* работает для Function Declaration. Для Function Declaration синтаксис не предусматривает возможность объявить дополнительное

«внутреннее» имя.

Зачастую, когда нам нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

Итого

Функции – это объекты.

Их свойства:

- `name` – имя функции. Обычно берётся из объявления функции, но если там нет – JavaScript пытается понять его из контекста.
- `length` – количество аргументов в объявлении функции. Троеточие («остаточные параметры») не считается.

Если функция объявлена как Function Expression (вне основного потока кода) и имеет имя, тогда это называется Named Function Expression (Именованным Функциональным Выражением). Это имя может быть использовано для ссылки на себя же, для рекурсивных вызовов и т.п.

Также функции могут содержать дополнительные свойства. Многие известные JavaScript-библиотеки искусно используют эту возможность.

Они создают «основную» функцию и добавляют множество «вспомогательных» функций внутрь первой. Например, библиотека jQuery создаёт функцию с именем `$`. Библиотека lodash создаёт функцию `_`, а потом добавляет в неё `_.clone`, `_.keyBy` и другие свойства (чтобы узнать о ней побольше см. документацию). Они делают это, чтобы уменьшить засорение глобального пространства имён посредством того, что одна библиотека

предоставляет только одну глобальную переменную, уменьшая вероятность конфликта имён.

Таким образом, функция может не только делать что-то сама по себе, но также и предоставлять полезную функциональность через свои свойства.

Задачи

Установка и уменьшение значения счётчика

важность: 5

Измените код `makeCounter()` так, чтобы счётчик мог увеличивать и устанавливать значение:

- `counter()` должен возвращать следующее значение (как и раньше).
- `counter.set(value)` должен устанавливать счётчику значение `value`.
- `counter.decrease()` должен уменьшать значение счётчика на 1.

Посмотрите код из песочницы с полным примером использования.

P.S. Для того, чтобы сохранить текущее значение счётчика, можно воспользоваться как замыканием, так и свойством функции. Или сделать два варианта решения: и так, и так.

Открыть песочницу с тестами для задачи.

К решению

Сумма с произвольным количеством скобок

важность: 2

Напишите функцию `sum`, которая бы работала следующим образом:

```
sum(1)(2) == 3; // 1 + 2
sum(1)(2)(3) == 6; // 1 + 2 + 3
sum(5)(-1)(2) == 6
sum(6)(-1)(-2)(-3) == 0
sum(0)(1)(2)(3)(4)(5) == 15
```

P.S. Подсказка: возможно вам стоит сделать особый метод преобразования в примитив для функции.

К решению

Синтаксис "new Function"

Существует ещё один вариант объявлять функции. Он используется крайне редко, но иногда другого решения не найти.

Синтаксис

Синтаксис для объявления функции:

```
let func = new Function([arg1, arg2, ...argN], functionBody);
```

Функция создаётся с заданными аргументами `arg1...argN` и телом `functionBody`.

Это проще понять на конкретном примере. Здесь объявлена функция с двумя аргументами:

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3
```

А вот функция без аргументов, в этом случае достаточно указать только тело:

```
let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

Главное отличие от других способов объявления функции, которые были рассмотрены ранее, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения.

Все предыдущие объявления требовали от нас, программистов, писать объявление функции в скрипте.

Но `new Function` позволяет превратить любую строку в функцию. Например, можно получить новую функцию с сервера и затем выполнить её:

```
let str = ... код, полученный с сервера динамически ...

let func = new Function(str);
func();
```

Это используется в очень специфических случаях, например, когда мы получаем код с сервера для динамической компиляции функции из шаблона, в сложных веб-приложениях.

Замыкание

Обычно функция запоминает, где родилась, в специальном свойстве `[[Environment]]`. Это ссылка на лексическое окружение (Lexical Environment), в котором она создана (мы разбирали это в главе Замыкание).

Но когда функция создаётся с использованием `new Function`, в её `[[Environment]]` записывается ссылка не на внешнее лексическое окружение, в котором она была создана, а на глобальное. Поэтому такая функция имеет доступ только к глобальным переменным.

```
function getFunc() {
  let value = "test";



  return func;
}

getFunc()(); // ошибка: value не определено
   let func = new Function('alert(value)');
```

Сравним это с обычным объявлением:

```
function getFunc() {
  let value = "test";



  return func;
}

getFunc()(); // , из лексического окружения функции getFunc
   let func = function() { alert(value); };
"test"
```

Эта особенность `new Function` выглядит странно, но оказывается очень полезной на практике.

Представьте, что нужно создать функцию из строки. Код

этой функции неизвестен во время написания скрипта (поэтому не используем обычные функции), а будет определён только в процессе выполнения. Мы можем получить код с сервера или с другого ресурса.

Наша новая функция должна взаимодействовать с основным скриптом.

Что если бы она имела доступ к внешним переменным?

Проблема в том, что перед отправкой JavaScript-кода на реальные работающие проекты код сжимается с помощью *минификатора* – специальной программы, которая уменьшает размер кода, удаляя комментарии, лишние пробелы, и, что самое главное, локальным переменным даются укороченные имена.

Например, если в функции объявляется переменная `let userName`, то минификатор изменяет её на `let a` (или другую букву, если она не занята) и изменяет её везде. Обычно так делать безопасно, потому что переменная является локальной, и никто снаружи не имеет к ней доступ. И внутри функции минификатор заменяет каждое её упоминание. Минификаторы достаточно умные. Они не просто осуществляют «тупой» поиск-замену, они анализируют структуру кода, и поэтому ничего не ломается.

Так что если бы даже `new Function` и имела доступ к внешним переменным, она не смогла бы найти переименованную `userName`.

Если бы `new Function` имела доступ к внешним переменным, при этом были бы проблемы с минификаторами.

Кроме того, такой код был бы архитектурно хуже и более подвержен ошибкам.

Чтобы передать что-то в функцию, созданную как `new Function`, можно использовать её аргументы.

Итого

Синтаксис:

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

По историческим причинам аргументы также могут быть объявлены через запятую в одной строке.

Эти 3 объявления ниже эквивалентны:

```
new Function('a', 'b', 'return a + b'); // стандартный синтаксис
new Function('a,b', 'return a + b'); // через запятую в одной строке
new Function('a , b', 'return a + b'); // через запятую с пробелами в одной строке
```

Функции, объявленные через `new Function`, имеют `[[Environment]]`, ссылающийся на глобальное лексическое окружение, а не на родительское. Поэтому они не могут использовать внешние локальные переменные. Но это очень хорошо, потому что страхует нас от ошибок. Переданные явно параметры – гораздо лучшее архитектурное решение, которое не вызывает проблем у минификаторов.

Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

`setTimeout` позволяет вызвать функцию **один раз** через определённый интервал времени.

`setInterval` позволяет вызывать функцию **регулярно**, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

setTimeout

Синтаксис:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Параметры:

`func|code`

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

`delay`

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

`arg1, arg2...`

Аргументы, передаваемые в функцию (не поддерживается в IE9-)

Например, данный код вызывает `sayHi()` спустя одну секунду:

```
function sayHi() {
  alert('Привет');
}

setTimeout(sayHi, 1000);
```

С аргументами:

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон
```

Если первый аргумент является строкой, то JavaScript создаст из неё функцию.

Это также будет работать:

```
setTimeout("alert('Привет')", 1000);
```

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

```
setTimeout(() => alert('Привет'), 1000);
```

Передавайте функцию, но не запускайте её

Начинающие разработчики иногда ошибаются, добавляя скобки `()` после функции:

```
// не правильно!
setTimeout(sayHi(), 1000);
```

Это не работает, потому что `setTimeout` ожидает ссылку на функцию. Здесь `sayHi()` запускает выполнение функции, и *результат выполнения* отправляется в `setTimeout`. В нашем случае результатом выполнения `sayHi()` является `undefined` (так как функция ничего не возвращает), поэтому ничего не планируется.

Отмена через clearTimeout

Вызов `setTimeout` возвращает «идентификатор таймера» `timerId`, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

В коде ниже планируем вызов функции и затем отменяем его (просто передумали). В результате ничего не происходит:

```
let timerId = setTimeout(() => alert("ничего не происходит"),
1000);
alert(timerId); // идентификатор таймера

clearTimeout(timerId);
alert(timerId); // тот же идентификатор (не принимает значение
null после отмены)
```

Как мы видим из вывода `alert`, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, Node.js возвращает объект таймера с дополнительными методами.

Повторюсь, что нет единой спецификации на эти методы, поэтому такое поведение является нормальным.

Для браузеров таймеры описаны в разделе таймеров стандарта HTML5.

setInterval

Метод `setInterval` имеет такой же синтаксис как `setTimeout`:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

Все аргументы имеют такое же значение. Но отличие этого метода от `setTimeout` в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать `clearInterval(timerId)`.

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

```
// повторить с интервалом 2 секунды
let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Во время показа `alert` время тоже идёт

В большинстве браузеров, включая Chrome и Firefox, внутренний счётчик продолжает тикать во время показа `alert/confirm/prompt`.

Так что если вы запустите код выше и подождёте с закрытием `alert` несколько секунд, то

следующий `alert` будет показан сразу, как только вы закроете предыдущий. Интервал времени между сообщениями `alert` будет короче, чем 2 секунды.

Рекурсивный setTimeout

Есть два способа запускать что-то регулярно.

Один из них `setInterval`. Другим является рекурсивный `setTimeout`. Например:

```
/** вместо:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');

}, 2000);
  timerId = setTimeout(tick, 2000); // (*)
```

Метод `setTimeout` выше планирует следующий вызов прямо после окончания текущего `(*)`.

Рекурсивный `setTimeout` – более гибкий метод, чем `setInterval`. С его помощью последующий вызов может быть задан по-разному в зависимости от результатов предыдущего.

Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд… Вот псевдокод:

```
let delay = 5000;

let timerId = setTimeout(function request() {
  ...отправить запрос...
```

```
  if (ошибка запроса из-за перегрузки сервера) {
    // увеличить интервал для следующего запроса
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

А если функции, которые мы планируем, ресурсоёмкие и требуют времени, то мы можем измерить время, затраченное на выполнение, и спланировать следующий вызов раньше или позже.

Рекурсивный `setTimeout` позволяет задать задержку между выполнениями более точно, чем `setInterval`.

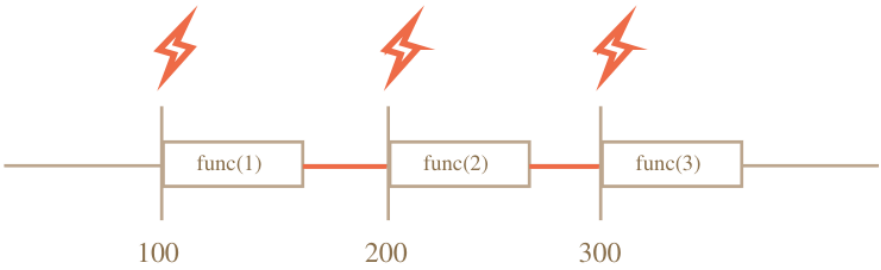
Сравним два фрагмента кода. Первый использует `setInterval`:

```
let i = 1;
setInterval(function() {
  func(i);
}, 100);
```

Второй использует рекурсивный `setTimeout`:

```
let i = 1;
setTimeout(function run() {
  func(i);
  setTimeout(run, 100);
}, 100);
```

Для `setInterval` внутренний планировщик будет выполнять `func(i)` каждые 100 мс:



Обратили внимание?

Реальная задержка между вызовами `func` с помощью `setInterval` меньше, чем указано в коде!

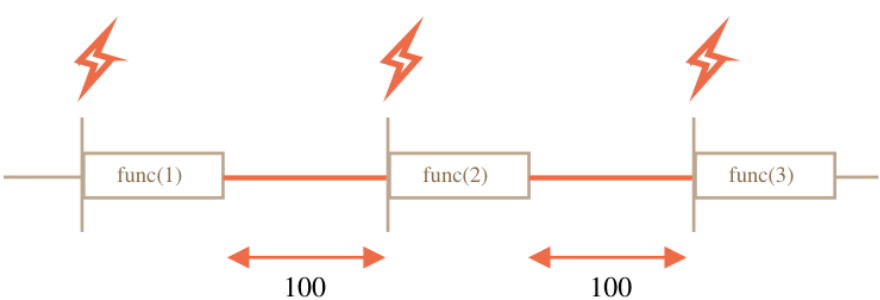
Это нормально, потому что время, затраченное на выполнение `func`, использует часть заданного интервала времени.

Вполне возможно, что выполнение `func` будет дольше, чем мы ожидали, и займёт более 100 мс.

В данном случае движок ждёт окончания выполнения `func` и затем проверяет планировщик и, если время истекло, *немедленно* запускает его снова.

В крайнем случае, если функция всегда выполняется дольше, чем задержка `delay`, то вызовы будут выполняться без задержек вообще.

Ниже представлено изображение, показывающее процесс работы рекурсивного `setTimeout`:



Рекурсивный `setTimeout` гарантирует фиксированную

задержку (здесь 100 мс).

Это потому, что новый вызов планируется в конце предыдущего.

Сборка мусора и колбэк setTimeout/setInterval

Когда функция передаётся в `setInterval/setTimeout`, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик мусора, даже если на неё нет других ссылок.

```
// функция остаётся в памяти до тех пор, пока планировщик обращается к ней
setTimeout(function() {...}, 100);
```

Для `setInterval` функция остаётся в памяти до тех пор, пока не будет вызван `clearInterval`.

Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

setTimeout с нулевой задержкой

Особый вариант использования: `setTimeout(func, 0)` или просто `setTimeout(func)`.

Это планирует вызов `func` настолько быстро, насколько

это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода.

Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

```
setTimeout(() => alert("Мир"));

alert("Привет");
```

Первая строка помещает вызов в «календарь» через 0 мс. Но планировщик проверит «календарь» только после того, как текущий код завершится. Поэтому `"Привет"` выводится первым, а `"Мир"` – после него.

Есть и более продвинутые случаи использования нулевой задержки в браузерах, которые мы рассмотрим в главе Событийный цикл: микрозадачи и макрозадачи.

Минимальная задержка вложенных таймеров в браузере

В браузере есть ограничение на то, как часто внутренние счётчики могут выполняться. В стандарте HTML5 говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.».

Продемонстрируем в примере ниже, что это означает. Вызов `setTimeout` повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время от предыдущего вызова в массиве `times`. Какова реальная задержка? Посмотрим:

```
let start = Date.now();
let times = [];

setTimeout(function run() {
  times.push(Date.now() - start); // запоминаем
задержку от предыдущего вызова

  if (start + 100 < Date.now()) alert(times); //
показываем задержку через 100 мс
  else setTimeout(run); // если нужно ещё
запланировать
});

// пример вывода:
//
1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,1
```

Первый таймер запускается сразу (как и указано в спецификации), а затем задержка вступает в игру, и мы видим `9, 15, 20, 24...`.

Аналогичное происходит при использовании `setInterval` вместо `setTimeout`: `setInterval(f)` запускает `f` несколько раз с нулевой задержкой, а затем с задержкой 4+ мс.

Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам.

Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, setImmediate для Node.js. Так что это ограничение относится только к браузерам.

Итого

Методы `setInterval(func, delay, ...args)` и `setTimeout(func, delay, ...args)` позволяют выполнять `func` регулярно или только один раз после

задержки `delay`, заданной в мс.

Для отмены выполнения необходимо вызвать `clearInterval/clearTimeout` со значением, которое возвращают методы `setInterval/setTimeout`.

Вложенный вызов `setTimeout` является более гибкой альтернативой `setInterval`. Также он позволяет более точно задать интервал между выполнениями.

Планирование с нулевой задержкой `setTimeout(func,0)` или, что то же самое, `setTimeout(func)` используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.

Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами `setTimeout`, а также для `setInterval`, начиная с 5-го вызова.

Обратим внимание, что все методы планирования *не гарантируют* точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

Перегружен процессор.

Вкладка браузера в фоновом режиме.

Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

Задачи

Вывод каждую секунду

важность: 5

Напишите функцию `printNumbers(from, to)`, которая выводит число каждую секунду, начиная от `from` и заканчивая `to`.

Сделайте два варианта решения.

1. Используя `setInterval`.
2. Используя рекурсивный `setTimeout`.

К решению

Что покажет setTimeout?

важность: 5

В приведённом ниже коде запланирован вызов `setTimeout`, а затем выполняется сложное вычисление, для завершения которого требуется более 100 мс.

Когда будет выполнена запланированная функция?

1. После цикла.
2. Перед циклом.
3. В начале цикла.

Что покажет `alert`?

```
let i = 0;

setTimeout(() => alert(i), 100); // ?
```

```
// предположим, что время выполнения этой функции >100 мс
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

К решению

Декораторы и переадресация вызова, call/apply

JavaScript предоставляет исключительно гибкие возможности по работе с функциями: они могут быть переданы в другие функции, использованы как объекты, и сейчас мы рассмотрим, как *перенаправлять* вызовы между ними и как их декорировать.

Прозрачное кеширование

Представим, что у нас есть функция `slow(x)`, выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же `x` она всегда возвращает один и тот же результат.

Если функция вызывается часто, то, вероятно, мы захотим кешировать (запоминать) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять `slow(x)` дополнительной функциональностью, мы заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Далее мы увидим, что в таком подходе масса преимуществ.

Вот код с объяснениями:

```
function slow(x) {
  // здесь могут быть ресурсоёмкие вычисления
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
```

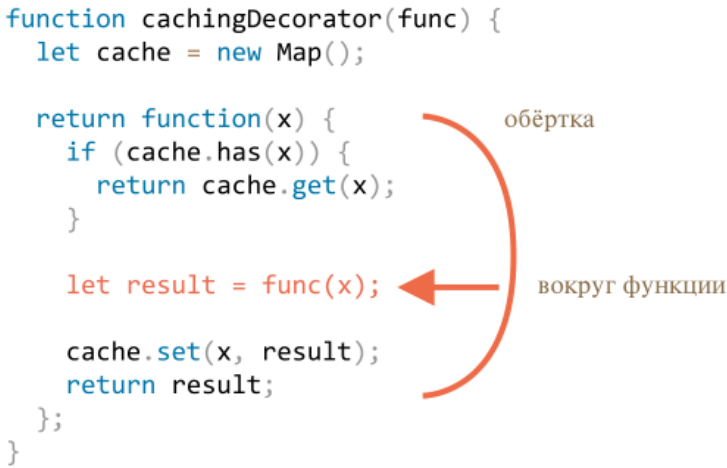
```
    if (cache.has(x)) {     // если кеш содержит такой x,
      return cache.get(x); // читаем из него результат
    }

    let result = func(x); // иначе, вызываем функцию

    cache.set(x, result); // и кешируем (запоминаем) результат
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем
alert( "Again: " + slow(1) ); // возвращаем из кеша

alert( slow(2) ); // slow(2) кешируем
alert( "Again: " + slow(2) ); // возвращаем из кеша
```

В коде выше `cachingDecorator` – это *декоратор*, специальная функция, которая принимает другую функцию и изменяет её поведение.

Идея состоит в том, что мы можем вызвать `cachingDecorator` с любой функцией, в результате чего мы получим кеширующую обёртку. Это здорово, т.к. у нас может быть множество функций, использующих такую функциональность, и всё, что нам нужно сделать – это применить к ним `cachingDecorator`.

Отделяя кеширующий код от основного кода, мы также сохраняем чистоту и простоту последнего.

Результат вызова `cachingDecorator(func)` является «обёрткой», т.е. `function(x)` «оборачивает» вызов `func(x)` в кеширующую логику:



С точки зрения внешнего кода, обёрнутая функция `slow` по-прежнему делает то же самое. Обёртка всего лишь добавляет к её поведению аспект кеширования.

Подводя итог, можно выделить несколько преимуществ использования отдельной `cachingDecorator` вместо изменения кода самой `slow`:

- Функцию `cachingDecorator` можно использовать повторно. Мы можем применить её к другой функции.
- Логика кеширования является отдельной, она не увеличивает сложность самой `slow` (если таковая была).
- При необходимости мы можем объединить несколько декораторов (речь об этом пойдёт позже).

Применение «func.call» для передачи контекста.

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов.

Например, в приведённом ниже коде `worker.slow()` перестаёт работать после применения декоратора:

```
// сделаем worker.slow кеширующим
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // здесь может быть страшно тяжёлая задача для процессора
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

// тот же код, что и выше
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // оригинальный метод работает

worker.slow = cachingDecorator(worker.slow); // теперь сделаем его кеширующим

    let result = func(x); // (**)
alert( worker.slow(2) ); // Ой! Ошибка: не удаётся прочитать свойство 'someMethod' из 'undefined'
```

Ошибка возникает в строке `(*)`. Функция пытается получить доступ к `this.someMethod` и завершается с ошибкой. Видите почему?

Причина в том, что в строке `(**)` декоратор вызывает оригинальную функцию как `func(x)`, и она в данном случае получает `this = undefined`.

Мы бы наблюдали похожую ситуацию, если бы попытались запустить:

```
let func = worker.slow;
func(2);
```

Т.е. декоратор передаёт вызов оригинальному методу, но без контекста. Следовательно – ошибка.

Давайте это исправим.

Существует специальный встроенный метод функции func.call(context, …args), который позволяет вызывать функцию, явно устанавливая `this`.

Синтаксис:

```
func.call(context, arg1, arg2, ...)
```

Он запускает функцию `func`, используя первый аргумент как её контекст `this`, а последующие – как её аргументы.

Проще говоря, эти два вызова делают почти то же самое:

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

Они оба вызывают `func` с аргументами `1`, `2` и `3`. Единственное отличие состоит в том, что `func.call` ещё и устанавливает `this` равным `obj`.

Например, в приведённом ниже коде мы вызываем `sayHi` в контексте различных объектов: `sayHi.call(user)` запускает `sayHi`, передавая `this=user`, а следующая строка устанавливает `this=admin`:

```
function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// используем 'call' для передачи различных объектов в
```

```
качестве 'this'
sayHi.call( user ); // John
sayHi.call( admin ); // Admin
```

Здесь мы используем `call` для вызова `say` с заданным контекстом и фразой:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// 'user' становится 'this', и "Hello" становится первым
аргументом
say.call( user, "Hello" ); // John: Hello
```

В нашем случае мы можем использовать `call` в обёртке для передачи контекста в исходную функцию:

```
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow); // теперь сделаем
её кеширующей

alert( worker.slow(2) ); // работает
```

```
alert( worker.slow(2) ); // работает, не вызывая первоначальную функцию (кешируется)
    let result = func.call(this, x); // теперь 'this' передаётся правильно
```

Теперь всё в порядке.

Чтобы всё было понятно, давайте посмотрим глубже, как передаётся `this`:

1. После *декорации* `worker.slow` становится обёрткой `function (x) { ... }`.
2. Так что при выполнении `worker.slow(2)` обёртка получает `2` в качестве аргумента и `this=worker` (так как это объект перед точкой).
3. Внутри обёртки, если результат ещё не кеширован, `func.call(this, x)` передаёт текущий `this` (`=worker`) и текущий аргумент (`=2`) в оригинальную функцию.

Переходим к нескольким аргументам с «func.apply»

Теперь давайте сделаем `cachingDecorator` ещё более универсальным. До сих пор он работал только с функциями с одним аргументом.

Как же кешировать метод с несколькими аргументами `worker.slow`?

```
let worker = {
  slow(min, max) {
    return min + max; // здесь может быть тяжёлая задача
  }
};

// будет кешировать вызовы с одинаковыми аргументами
worker.slow = cachingDecorator(worker.slow);
```

Здесь у нас есть две задачи для решения.

Во-первых, как использовать оба аргумента `min` и `max` для ключа в коллекции `cache`? Ранее для одного аргумента `x` мы могли просто сохранить результат `cache.set(x, result)` и вызвать `cache.get(x)`, чтобы получить его позже. Но теперь нам нужно запомнить результат для *комбинации аргументов* `(min,max)`. Встроенный `Map` принимает только одно значение как ключ.

Есть много возможных решений:

1. Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна, чем встроенный `Map`, и поддерживает множественные ключи.
2. Использовать вложенные коллекции: `cache.set(min)` будет `Map`, которая хранит пару `(max, result)`. Тогда получить `result` мы сможем, вызвав `cache.get(min).get(max)`.
3. Соединить два значения в одно. В нашем конкретном случае мы можем просто использовать строку `"min,max"` как ключ к `Map`. Для гибкости, мы можем позволить передавать *хеширующую функцию* в декоратор, которая знает, как сделать одно значение из многих.

Для многих практических применений третий вариант достаточно хорош, поэтому мы будем придерживаться его.

Также нам понадобится заменить `func.call(this, x)` на `func.call(this, ...arguments)`, чтобы передавать все аргументы обёрнутой функции, а не только первый.

Вот более мощный `cachingDecorator`:

```
let worker = {
```

```
  slow(min, max) {
    alert(`Called with ${min},${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {

    if (cache.has(key)) {
      return cache.get(key);
    }



    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // работает
alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)
    let key = hash(arguments); // (*)
    let result = func.call(this,
...arguments); // (**)
```

Теперь он работает с любым количеством аргументов.

Есть два изменения:

- В строке `(*)` вызываем `hash` для создания одного ключа из `arguments`. Здесь мы используем простую функцию «объединения», которая превращает аргументы `(3, 5)` в ключ `"3,5"`. В более сложных случаях могут потребоваться другие функции хеширования.

  Затем в строке `(**)` используем `func.call(this, ...arguments)` для передачи как контекста, так и всех аргументов, полученных обёрткой (независимо от их

количества), в исходную функцию.

Вместо `func.call(this, ...arguments)` мы могли бы написать `func.apply(this, arguments)`.

Синтаксис встроенного метода func.apply:

```
func.apply(context, args)
```

Он выполняет `func`, устанавливая `this=context` и принимая в качестве списка аргументов псевдомассив `args`.

Единственная разница в синтаксисе между `call` и `apply` состоит в том, что `call` ожидает список аргументов, в то время как `apply` принимает псевдомассив.

Эти два вызова почти эквивалентны:

```
func.call(context, ...args); // передаёт массив как список с оператором расширения
func.apply(context, args);   // тот же эффект
```

Есть только одна небольшая разница:

> Оператор расширения `...` позволяет передавать *перебираемый* объект `args` в виде списка в `call`.
>
> А `apply` принимает только *псевдомассив* `args`.

Так что эти вызовы дополняют друг друга. Для перебираемых объектов сработает `call`, а где мы ожидаем псевдомассив – `apply`.

А если у нас объект, который и то, и другое, например, реальный массив, то технически мы могли бы использовать любой метод, но `apply`, вероятно, будет быстрее, потому что большинство движков JavaScript внутренне оптимизируют его лучше.

Передача всех аргументов вместе с контекстом другой функции называется «перенаправлением вызова» (call forwarding).

Простейший вид такого перенаправления:

```
let wrapper = function() {
  return func.apply(this, arguments);
};
```

При вызове `wrapper` из внешнего кода его не отличить от вызова исходной функции.

Заимствование метода

Теперь давайте сделаем ещё одно небольшое улучшение функции хеширования:

```
function hash(args) {
  return args[0] + ',' + args[1];
}
```

На данный момент она работает только для двух аргументов. Было бы лучше, если бы она могла склеить любое количество `args`.

Естественным решением было бы использовать метод arr.join:

```
function hash(args) {
  return args.join();
}
```

…К сожалению, это не сработает, потому что мы вызываем `hash(arguments)`, а объект `arguments` является перебираемым и псевдомассивом, но не реальным массивом.

Таким образом, вызов `join` для него потерпит неудачу, что мы и можем видеть ниже:

```
function hash() {

}

hash(1, 2);
  alert( arguments.join() ); // Ошибка:
arguments.join не является функцией
```

Тем не менее, есть простой способ использовать соединение массива:

```
function hash() {

}

hash(1, 2);
  alert( [].join.call(arguments) ); // 1,2
```

Этот трюк называется *заимствование метода*.

Мы берём (заимствуем) метод `join` из обычного массива `[].join`. И используем `[].join.call`, чтобы выполнить его в контексте `arguments`.

Почему это работает?

Это связано с тем, что внутренний алгоритм встроенного метода `arr.join(glue)` очень прост. Взято из спецификации практически «как есть»:

1. Пускай первым аргументом будет `glue` или, в случае отсутствия аргументов, им будет запятая `","`
2. Пускай `result` будет пустой строкой `""`.
3. Добавить `this[0]` к `result`.
4. Добавить `glue` и `this[1]`.
5. Добавить `glue` и `this[2]`.

6. …выполнять до тех пор, пока `this.length` элементов не будет склеено.
7. Вернуть `result`.

Таким образом, технически он принимает `this` и объединяет `this[0], this[1]`… и т.д. вместе. Он намеренно написан так, что допускает любой псевдомассив `this` (не случайно, многие методы следуют этой практике). Вот почему он также работает с `this=arguments`.

Итого

Декоратор – это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией.

Обычно безопасно заменить функцию или метод декорированным, за исключением одной мелочи. Если исходная функция предоставляет свойства, такие как `func.calledCount` или типа того, то декорированная функция их не предоставит. Потому что это обёртка. Так что нужно быть осторожным в их использовании. Некоторые декораторы предоставляют свои собственные свойства.

Декораторы можно рассматривать как «дополнительные возможности» или «аспекты», которые можно добавить в функцию. Мы можем добавить один или несколько декораторов. И всё это без изменения кода оригинальной функции!

Для реализации `cachingDecorator` мы изучили методы:

func.call(context, arg1, arg2…) – вызывает `func` с данным контекстом и аргументами.

func.apply(context, args) – вызывает `func`, передавая

`context` как `this` и псевдомассив `args` как список аргументов.

В основном *переадресация вызова* выполняется с помощью `apply`:

```
let wrapper = function(original, arguments) {
  return original.apply(this, arguments);
};
```

Мы также рассмотрели пример *заимствования метода*, когда мы вызываем метод у объекта в контексте другого объекта. Весьма распространено заимствовать методы массива и применять их к `arguments`. В качестве альтернативы можно использовать объект с остаточными параметрами `...args`, который является реальным массивом.

На практике декораторы используются для самых разных задач. Проверьте, насколько хорошо вы их освоили, решая задачи этой главы.

Задачи

Декоратор-шпион

важность: 5

Создайте декоратор `spy(func)`, который должен возвращать обёртку, которая сохраняет все вызовы функции в своём свойстве `calls`.

Каждый вызов должен сохраняться как массив аргументов.

Например:

```
function work(a, b) {
  alert( a + b ); // произвольная функция или метод
}



work(1, 2); // 3
work(4, 5); // 9

for (let args of work.calls) {
  alert( 'call:' + args.join() ); // "call:1,2", "call:4,5"
}
work = spy(work);
```

P.S.: Этот декоратор иногда полезен для юнит-тестирования. Его расширенная форма – `sinon.spy` – содержится в библиотеке Sinon.JS.

Открыть песочницу с тестами для задачи.

К решению

Задерживающий декоратор

важность: 5

Создайте декоратор `delay(f, ms)`, который задерживает каждый вызов `f` на `ms` миллисекунд. Например:

```
function f(x) {
  alert(x);
}

// создаём обёртки
let f1000 = delay(f, 1000);
let f1500 = delay(f, 1500);

f1000("test"); // показывает "test" после 1000 мс
f1500("test"); // показывает "test" после 1500 мс
```

Другими словами, `delay(f, ms)` возвращает вариант `f`

с «задержкой на `ms` мс».

В приведённом выше коде `f` – функция с одним аргументом, но ваше решение должно передавать все аргументы и контекст `this`.

Открыть песочницу с тестами для задачи.

К решению

Декоратор debounce

важность: 5

Результатом декоратора `debounce(f, ms)` должна быть обёртка, которая передаёт вызов `f` не более одного раза в `ms` миллисекунд. Другими словами, когда мы вызываем `debounce`, это гарантирует, что все остальные вызовы будут игнорироваться в течение `ms`.

Например:

```
let f = debounce(alert, 1000);

f(1); // выполняется немедленно
f(2); // проигнорирован

setTimeout( () => f(3), 100); // проигнорирован (прошло только 100 мс)
setTimeout( () => f(4), 1100); // выполняется
setTimeout( () => f(5), 1500); // проигнорирован (прошло только 400 мс от последнего вызова)
```

На практике `debounce` полезен для функций, которые получают/обновляют данные, и мы знаем, что повторный вызов в течение короткого промежутка времени не даст ничего нового. Так что лучше не тратить на него ресурсы.

Открыть песочницу с тестами для задачи.

К решению

Тормозящий (throttling) декоратор

важность: 5

Создайте «тормозящий» декоратор `throttle(f, ms)`, который возвращает обёртку, передавая вызов в `f` не более одного раза в `ms` миллисекунд. Те вызовы, которые попадают в период «торможения», игнорируются.

Отличие от `debounce` – если проигнорированный вызов является последним во время «задержки», то он выполняется в конце.

Давайте рассмотрим реальное применение, чтобы лучше понять это требование и выяснить, откуда оно взято.

Например, мы хотим отслеживать движения мыши.

В браузере мы можем объявить функцию, которая будет запускаться при каждом движении указателя и получать его местоположение. Во время активного использования мыши эта функция запускается очень часто, это может происходить около 100 раз в секунду (каждые 10 мс).

Мы бы хотели обновлять информацию на странице при передвижениях.

…Но функция обновления `update()` слишком ресурсоёмкая, чтобы делать это при каждом микродвижении. Да и нет смысла делать обновление

чаще, чем один раз в 1000 мс.

Поэтому мы обернём вызов в декоратор: будем использовать `throttle(update, 1000)` как функцию, которая будет запускаться при каждом перемещении указателя вместо оригинальной `update()`. Декоратор будет вызываться часто, но передавать вызов в `update()` максимум раз в 1000 мс.

Визуально это будет выглядеть вот так:

1. Для первого движения указателя декорированный вариант сразу передаёт вызов в `update`. Это важно, т.к. пользователь сразу видит нашу реакцию на его перемещение.
2. Затем, когда указатель продолжает движение, в течение 1000 мс ничего не происходит. Декорированный вариант игнорирует вызовы.
3. По истечению 1000 мс происходит ещё один вызов `update` с последними координатами.
4. Затем, наконец, указатель где-то останавливается. Декорированный вариант ждёт, пока не истечёт 1000 мс, и затем вызывает `update` с последними координатами. В итоге окончательные координаты указателя тоже обработаны.

Пример кода:

```
function f(a) {
  console.log(a)
}

// f1000 передаёт вызовы f максимум раз в 1000 мс
let f1000 = throttle(f, 1000);

f1000(1); // показывает 1
```

```
f1000(2); // (ограничение, 1000 мс ещё нет)
f1000(3); // (ограничение, 1000 мс ещё нет)

// когда 1000 мс истекли ...
// ...выводим 3, промежуточное значение 2 было проигнорировано
```

P.S. Аргументы и контекст `this`, переданные в `f1000`, должны быть переданы в оригинальную `f`.

Открыть песочницу с тестами для задачи.

К решению

Привязка контекста к функции

При передаче методов объекта в качестве колбэков, например для `setTimeout`, возникает известная проблема – потеря `this`.

В этой главе мы посмотрим, как её можно решить.

Потеря «this»

Мы уже видели примеры потери `this`. Как только метод передаётся отдельно от объекта – `this` теряется.

Вот как это может произойти в случае с `setTimeout`:

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};
```

```
setTimeout(user.sayHi, 1000); // Привет, undefined!
```

При запуске этого кода мы видим, что вызов `this.firstName` возвращает не «Вася», а `undefined`!

Это произошло потому, что `setTimeout` получил функцию `sayHi` отдельно от объекта `user` (именно здесь функция и потеряла контекст). То есть последняя строка может быть переписана как:

```
let f = user.sayHi;
setTimeout(f, 1000); // контекст user потеряли
```

Метод `setTimeout` в браузере имеет особенность: он устанавливает `this=window` для вызова функции (в Node.js `this` становится объектом таймера, но здесь это не имеет значения). Таким образом, для `this.firstName` он пытается получить `window.firstName`, которого не существует. В других подобных случаях `this` обычно просто становится `undefined`.

Задача довольно типичная – мы хотим передать метод объекта куда-то ещё (в этом конкретном случае – в планировщик), где он будет вызван. Как бы сделать так, чтобы он вызывался в правильном контексте?

Решение 1: сделать функцию-обёртку

Самый простой вариант решения – это обернуть вызов в анонимную функцию, создав замыкание:

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
```

```
  }
};

setTimeout(function() {
  user.sayHi(); // Привет, Вася!
}, 1000);
```

Теперь код работает корректно, так как объект `user` достаётся из замыкания, а затем вызывается его метод `sayHi`.

То же самое, только короче:

```
setTimeout(() => user.sayHi(), 1000); // Привет, Вася!
```

Выглядит хорошо, но теперь в нашем коде появилась небольшая уязвимость.

Что произойдёт, если до момента срабатывания `setTimeout` (ведь задержка составляет целую секунду!) в переменную `user` будет записано другое значение? Тогда вызов неожиданно будет совсем не тот!

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};

setTimeout(() => user.sayHi(), 1000);

// ...в течение 1 секунды
user = { sayHi() { alert("Другой пользователь в
'setTimeout'!"); } };

// Другой пользователь в 'setTimeout'!
```

Следующее решение гарантирует, что такого не случится.

Решение 2: привязать контекст с помощью bind

В современном JavaScript у функций есть встроенный метод bind, который позволяет зафиксировать `this`.

Базовый синтаксис `bind`:

```
// полный синтаксис будет представлен немного позже
let boundFunc = func.bind(context);
```

Результатом вызова `func.bind(context)` является особый «экзотический объект» (термин взят из спецификации), который вызывается как функция и прозрачно передаёт вызов в `func`, при этом устанавливая `this=context`.

Другими словами, вызов `boundFunc` подобен вызову `func` с фиксированным `this`.

Например, здесь `funcUser` передаёт вызов в `func`, фиксируя `this=user`:

```
let user = {
  firstName: "Вася"
};

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // Вася
```

Здесь `func.bind(user)` — это «связанный вариант» `func`, с фиксированным `this=user`.

Все аргументы передаются исходному методу `func` как есть, например:

```
let user = {
  firstName: "Вася"
};
```

```
function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}

// привязка this к user
let funcUser = func.bind(user);

funcUser("Привет"); // Привет, Вася (аргумент
"Привет" передан, при этом this = user)
```

Теперь давайте попробуем с методом объекта:

```
let user = {
  firstName: "Вася",
  sayHi() {
    alert(`Привет, ${this.firstName}!`);
  }
};



sayHi(); // Привет, Вася!

setTimeout(sayHi, 1000); // Привет, Вася!
let sayHi = user.sayHi.bind(user); // (*)
```

В строке `(*)` мы берём метод `user.sayHi` и привязываем его к `user`. Теперь `sayHi` – это «связанная» функция, которая может быть вызвана отдельно или передана в `setTimeout` (контекст всегда будет правильным).

Здесь мы можем увидеть, что `bind` исправляет только `this`, а аргументы передаются как есть:

```
let user = {
  firstName: "Вася",
  say(phrase) {
    alert(`${phrase}, ${this.firstName}!`);
  }
};

let say = user.say.bind(user);

say("Привет"); // Привет, Вася (аргумент "Привет" передан в
```

```
функцию "say")
say("Пока"); // Пока, Вася (аргумент "Пока" передан в функцию
"say")
```

Удобный метод: `bindAll`

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле:

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например _.bindAll(obj) в lodash.

Частичное применение

До сих пор мы говорили только о привязывании `this`. Давайте шагнём дальше.

Мы можем привязать не только `this`, но и аргументы. Это делается редко, но иногда может быть полезно.

Полный синтаксис `bind`:

```
let bound = func.bind(context, [arg1], [arg2], ...);
```

Это позволяет привязать контекст `this` и начальные аргументы функции.

Например, у нас есть функция умножения `mul(a, b)`:

```
function mul(a, b) {
  return a * b;
}
```

Давайте воспользуемся `bind`, чтобы создать функцию `double` на её основе:

```
function mul(a, b) {
  return a * b;
}
```

```
alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
let double = mul.bind(null, 2);
```

Вызов `mul.bind(null, 2)` создаёт новую функцию `double`, которая передаёт вызов `mul`, фиксируя `null` как контекст, и `2` – как первый аргумент. Следующие аргументы передаются как есть.

Это называется частичное применение – мы создаём новую функцию, фиксируя некоторые из существующих параметров.

Обратите внимание, что в данном случае мы на самом деле не используем `this`. Но для `bind` это обязательный параметр, так что мы должны передать туда что-нибудь вроде `null`.

В следующем коде функция `triple` умножает значение на три:

```
function mul(a, b) {
  return a * b;
}
```

```
alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
let triple = mul.bind(null, 3);
```

Для чего мы обычно создаём частично применённую функцию?

Польза от этого в том, что возможно создать независимую функцию с понятным названием (`double`, `triple`). Мы можем использовать её и не передавать каждый раз первый аргумент, т.к. он зафиксирован с помощью `bind`.

В других случаях частичное применение полезно, когда у нас есть очень общая функция и для удобства мы хотим создать её более специализированный вариант.

Например, у нас есть функция `send(from, to, text)`. Потом внутри объекта `user` мы можем захотеть использовать её частный вариант: `sendTo(to, text)`, который отправляет текст от имени текущего пользователя.

Частичное применение без контекста

Что если мы хотим зафиксировать некоторые аргументы, но не контекст `this`? Например, для метода объекта.

Встроенный `bind` не позволяет этого. Мы не можем просто опустить контекст и перейти к аргументам.

К счастью, легко создать вспомогательную функцию `partial`, которая привязывает только аргументы.

Вот так:

```
// использование:
```

```
let user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}!`);
  }
};

// добавляем частично применённый метод с фиксированным временем
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello");
// Что-то вроде этого:
// [10:00] John: Hello!
function partial(func, ...argsBound) {
  return function(...args) { // (*)
    return func.call(this, ...argsBound, ...args);
  }
}
```

Результатом вызова `partial(func[, arg1, arg2...])` будет обёртка `(*)`, которая вызывает `func` с:

- Тем же `this`, который она получает (для вызова `user.sayNow` – это будет `user`)
- Затем передаёт ей `...argsBound` – аргументы из вызова `partial` (`"10:00"`)
- Затем передаёт ей `...args` – аргументы, полученные обёрткой (`"Hello"`)

Благодаря оператору расширения `...` реализовать это очень легко, не правда ли?

Также есть готовый вариант _.partial из библиотеки lodash.

Итого

Метод `bind` возвращает «привязанный вариант» функции

`func`, фиксируя контекст `this` и первые аргументы `arg1`, `arg2`..., если они заданы.

Обычно `bind` применяется для фиксации `this` в методе объекта, чтобы передать его в качестве колбэка. Например, для `setTimeout`.

Когда мы привязываем аргументы, такая функция называется «частично применённой» или «частичной».

Частичное применение удобно, когда мы не хотим повторять один и тот же аргумент много раз. Например, если у нас есть функция `send(from, to)` и `from` всё время будет одинаков для нашей задачи, то мы можем создать частично применённую функцию и дальше работать с ней.

Задачи

Связанная функция как метод

важность: 5

Что выведет функция?

```
function f() {
  alert( this ); // ?
}

let user = {
  g: f.bind(null)
};

user.g();
```

К решению

Повторный bind

важность: 5

Можем ли мы изменить `this` дополнительным связыванием?

Что выведет этот код?

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "Вася"} ).bind( {name: "Петя" } );

f();
```

К решению

Свойство функции после bind

важность: 5

В свойство функции записано значение. Изменится ли оно после применения `bind`? Обоснуйте ответ.

```
function sayHi() {
  alert( this.name );
}
sayHi.test = 5;

let bound = sayHi.bind({
  name: "Вася"
});

alert( bound.test ); // что выведет? почему?
```

К решению

Исправьте функцию, теряющую "this"

важность: 5

Вызов `askPassword()` в приведённом ниже коде должен проверить пароль и затем вызвать `user.loginOk/loginFail` в зависимости от ответа.

Однако, его вызов приводит к ошибке. Почему?

Исправьте выделенную строку, чтобы всё работало (других строк изменять не надо).

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'Вася',

  loginOk() {
    alert(`${this.name} logged in`);
  },

  loginFail() {
    alert(`${this.name} failed to log in`);
  },

};

askPassword(user.loginOk, user.loginFail);
```

К решению

Использование частично применённой функции для логина

важность: 5

Это задание является немного усложнённым вариантом одного из предыдущих – Исправьте функцию, теряющую

"this".

Объект `user` был изменён. Теперь вместо двух функций `loginOk/loginFail` у него есть только одна – `user.login(true/false)`.

Что нужно передать в вызов функции `askPassword` в коде ниже, чтобы она могла вызывать функцию `user.login(true)` как `ok` и функцию `user.login(false)` как `fail`?

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'John',

  login(result) {
    alert( this.name + (result ? ' logged in' : ' failed to
log in') );
  }
};

askPassword(?, ?); // ?
```

Ваши изменения должны затрагивать только выделенный фрагмент кода.

К решению

Повторяем стрелочные функции

Давайте вернёмся к стрелочным функциям.

Стрелочные функции – это не просто «сокращение», чтобы меньше писать. У них есть ряд других полезных особенностей.

При написании JavaScript-кода часто возникают ситуации, когда нам нужно написать небольшую функцию, которая будет выполнена где-то ещё.

Например:

- `arr.forEach(func)` – `func` выполняется `forEach` для каждого элемента массива.
- `setTimeout(func)` – `func` выполняется встроенным планировщиком.
- …и так далее.

Это очень в духе JavaScript – создать функцию и передать её куда-нибудь.

И в таких функциях мы обычно не хотим выходить из текущего контекста. Здесь как раз и полезны стрелочные функции.

У стрелочных функций нет «this»

Как мы помним из главы Методы объекта, "this", у стрелочных функций нет `this`. Если происходит обращение к `this`, его значение берётся снаружи.

Например, мы можем использовать это для итерации внутри метода объекта:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
```

```
  }
};

group.showList();
    this.students.forEach(
      student => alert(this.title + ': ' +
student)
    );
```

Здесь внутри `forEach` использована стрелочная функция, таким образом `this.title` в ней будет иметь точно такое же значение, как в методе `showList`: `group.title`.

Если бы мы использовали «обычную» функцию, была бы ошибка:

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {

  }
};

group.showList();
    this.students.forEach(function(student) {
      // Error: Cannot read property 'title'
of undefined
      alert(this.title + ': ' + student)
    });
```

Ошибка возникает потому, что `forEach` по умолчанию выполняет функции с `this`, равным `undefined`, и в итоге мы пытаемся обратиться к `undefined.title`.

Это не влияет на стрелочные функции, потому что у них просто нет `this`.

Стрелочные функции нельзя использовать с `new`

Отсутствие `this` естественным образом ведёт к другому ограничению: стрелочные функции не могут быть использованы как конструкторы. Они не могут быть вызваны с `new`.

Стрелочные функции VS bind

Существует тонкая разница между стрелочной функцией `=>` и обычной функцией, вызванной с `.bind(this)`:

- `.bind(this)` создаёт «связанную версию» функции.
- Стрелка `=>` ничего не привязывает. У функции просто нет `this`. При получении значения `this` – оно, как обычная переменная, берётся из внешнего лексического окружения.

Стрелочные функции не имеют «arguments»

У стрелочных функций также нет переменной `arguments`.

Это отлично подходит для декораторов, когда нам нужно пробросить вызов с текущими `this` и `arguments`.

Например, `defer(f, ms)` принимает функцию и возвращает обёртку над ней, которая откладывает вызов на `ms` миллисекунд:

```
function defer(f, ms) {
  return function() {
```

```
    setTimeout(() => f.apply(this, arguments), ms)
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // выводит "Hello, John" через 2
секунды
```

То же самое без стрелочной функции выглядело бы так:

```
function defer(f, ms) {
  return function(...args) {
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  };
}
```

Здесь мы были вынуждены создать дополнительные переменные `args` и `ctx`, чтобы функция внутри `setTimeout` могла получить их.

Итого

Стрелочные функции:

- Не имеют `this`.
- Не имеют `arguments`.
- Не могут быть вызваны с `new`.
- (У них также нет `super`, но мы про это не говорили. Про это будет в главе Наследование классов).

Всё это потому, что они предназначены для небольшого кода, который не имеет своего «контекста», выполняясь в текущем. И они отлично справляются с этой задачей!

Свойства объекта, их конфигурация

В этой секции мы вернёмся к объектам и изучим их свойства более глубоко.

Флаги и дескрипторы свойств

Как мы знаем, объекты могут содержать свойства.

До этого момента мы рассматривали свойство только как пару «ключ-значение». Но на самом деле свойство объекта гораздо мощнее и гибче.

В этой главе мы изучим дополнительные флаги конфигурации для свойств, а в следующей – увидим, как можно незаметно превратить их в специальные функции – геттеры и сеттеры.

Флаги свойств

Помимо значения **`value`**, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

`writable` – если `true`, свойство можно изменить, иначе оно только для чтения.

`enumerable` – если `true`, свойство перечисляется в циклах, в противном случае циклы его игнорируют.

`configurable` – если `true`, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Мы ещё не встречали эти атрибуты, потому что обычно

они скрыты. Когда мы создаём свойство «обычным способом», все они имеют значение `true`. Но мы можем изменить их в любое время.

Сначала посмотрим, как получить их текущие значения.

Метод Object.getOwnPropertyDescriptor позволяет получить *полную* информацию о свойстве.

Его синтаксис:

```
let descriptor = Object.getOwnPropertyDescriptor(obj,
propertyName);
```

`obj`

Объект, из которого мы получаем информацию.

`propertyName`

Имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

Например:

```
let user = {
  name: "John"
};

let descriptor = Object.getOwnPropertyDescriptor(user,
'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* дескриптор свойства:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
```

```
*/
```

Чтобы изменить флаги, мы можем использовать метод Object.defineProperty.

Его синтаксис:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj, propertyName

Объект и его свойство, для которого нужно применить дескриптор.

descriptor

Применяемый дескриптор.

Если свойство существует, `defineProperty` обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение `false`.

Например, здесь создаётся свойство `name`, все флаги которого имеют значение `false`:

```
let user = {};



let descriptor = Object.getOwnPropertyDescriptor(user,
'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "John",

}
 */
Object.defineProperty(user, "name", {
  value: "John"
```

```
});
  "writable": false,
  "enumerable": false,
  "configurable": false
```

Сравните это с предыдущим примером, в котором мы создали свойство `user.name` «обычным способом»: в этот раз все флаги имеют значение `false`. Если это не то, что нам нужно, надо присвоить им значения `true` в параметре `descriptor`.

Теперь давайте рассмотрим на примерах, что нам даёт использование флагов.

Только для чтения

Сделаем свойство `user.name` доступным только для чтения. Для этого изменим флаг `writable`:

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {

});
```

```
  writable: false
user.name = "Pete"; // Ошибка: Невозможно изменить доступное только для чтения свойство 'name'
```

Теперь никто не сможет изменить имя пользователя, если только не обновит соответствующий флаг новым вызовом `defineProperty`.

Ошибки появляются только в строгом режиме

В нестрогом режиме, без `use strict`, мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Вот тот же пример, но свойство создано «с нуля»:

```
let user = { };

Object.defineProperty(user, "name", {

});

alert(user.name); // John
user.name = "Pete"; // Ошибка
  value: "John",
  // для нового свойства необходимо явно
указывать все флаги, для которых значение true
  enumerable: true,
  configurable: true
```

Неперечислимое свойство

Теперь добавим собственный метод `toString` к объекту `user`.

Встроенный метод `toString` в объектах – неперечислимый, его не видно в цикле `for..in`. Но если мы напишем свой собственный метод `toString`, цикл `for..in` будет выводить его по умолчанию:

```
let user = {
  name: "John",
  toString() {
```

```
    return this.name;
  }
};

// По умолчанию оба свойства выведутся:
for (let key in user) alert(key); // name, toString
```

Если мы этого не хотим, можно установить для свойства `enumerable:false`. Тогда оно перестанет появляться в цикле `for..in` аналогично встроенному `toString`:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {

});


for (let key in user) alert(key); // name
  enumerable: false
// Теперь наше свойство toString пропало из
цикла:
```

Неперечислимые свойства также не возвращаются `Object.keys`:

```
alert(Object.keys(user)); // name
```

Неконфигурируемое свойство

Флаг неконфигурируемого свойства (`configurable:false`) иногда предустановлен для некоторых встроенных объектов и свойств.

Неконфигурируемое свойство не может быть удалено.

Например, свойство `Math.PI` – только для чтения, неперечислимое и неконфигурируемое:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

То есть программист не сможет изменить значение `Math.PI` или перезаписать его.

```
Math.PI = 3; // Ошибка

// delete Math.PI тоже не сработает
```

Определение свойства как неконфигурируемого – это дорога в один конец. Мы не сможем отменить это действие, потому что `defineProperty` не работает с неконфигурируемыми свойствами.

В коде ниже мы делаем `user.name` «навечно запечатанной» константой:

```
let user = { };

Object.defineProperty(user, "name", {
  value: "John",
  writable: false,
  configurable: false
});

// теперь невозможно изменить user.name или
его флаги
// всё это не будет работать:
//   user.name = "Pete"
//   delete user.name
```

```
//   defineProperty(user, "name", ...)
Object.defineProperty(user, "name", {writable:
true}); // Ошибка
```

Ошибки отображаются только в строгом режиме

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Метод Object.defineProperties

Существует метод Object.defineProperties(obj, descriptors), который позволяет определять множество свойств сразу.

Его синтаксис:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

Например:

```
Object.defineProperties(user, {
  name: { value: "John", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

Таким образом, мы можем определить множество свойств

одной операцией.

Object.getOwnPropertyDescriptors

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом Object.getOwnPropertyDescriptors(obj).

Вместе с `Object.defineProperties` этот метод можно использовать для клонирования объекта вместе с его флагами:

```
let clone = Object.defineProperties({},
Object.getOwnPropertyDescriptors(obj));
```

Обычно при клонировании объекта мы используем присваивание, чтобы скопировать его свойства:

```
for (let key in user) {
  clone[key] = user[key]
}
```

…Но это не копирует флаги. Так что если нам нужен клон «получше», предпочтительнее использовать `Object.defineProperties`.

Другое отличие в том, что `for..in` игнорирует символьные свойства, а `Object.getOwnPropertyDescriptors` возвращает дескрипторы *всех* свойств, включая свойства-символы.

Глобальное запечатывание объекта

Дескрипторы свойств работают на уровне конкретных свойств.

Но ещё есть методы, которые ограничивают доступ ко *всему* объекту:

Object.preventExtensions(obj)

Запрещает добавлять новые свойства в объект.

Object.seal(obj)

Запрещает добавлять/удалять свойства. Устанавливает `configurable: false` для всех существующих свойств.

Object.freeze(obj)

Запрещает добавлять/удалять/изменять свойства. Устанавливает `configurable: false, writable: false` для всех существующих свойств.

А также есть методы для их проверки:

Object.isExtensible(obj)

Возвращает `false`, если добавление свойств запрещено, иначе `true`.

Object.isSealed(obj)

Возвращает `true`, если добавление/удаление свойств запрещено и для всех существующих свойств установлено `configurable: false`.

Object.isFrozen(obj)

Возвращает `true`, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено `configurable: false, writable: false`.

На практике эти методы используются редко.

Свойства - геттеры и сеттеры

Есть два типа свойств объекта.

Первый тип это *свойства-данные (data properties)*. Мы уже знаем, как работать с ними. Все свойства, которые мы использовали до текущего момента, были свойствами-данными.

Второй тип свойств мы ещё не рассматривали. Это *свойства-аксессоры (accessor properties)*. По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

Геттеры и сеттеры

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются `get` и `set`:

```
let obj = {
   {
    // геттер, срабатывает при чтении obj.propName
  },

   {
    // сеттер, срабатывает при записи obj.propName = value
  }
};
get
 propName()
set
 propName(value)
```

Геттер срабатывает, когда `obj.propName` читается, сеттер – когда значение присваивается.

Например, у нас есть объект `user` со свойствами `name` и

surname:

```
let user = {
  name: "John",
  surname: "Smith"
};
```

А теперь добавим свойство объекта `fullName` для полного имени, которое в нашем случае `"John Smith"`. Само собой, мы не хотим дублировать уже имеющуюся информацию, так что реализуем его при помощи аксессора:

```
let user = {
  name: "John",
  surname: "Smith",


};

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
alert(user.fullName); // John Smith
```

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. Мы не *вызываем* `user.fullName` как функцию, а *читаем* как обычное свойство: геттер выполнит всю работу за кулисами.

```
let user = {
  get fullName() {
    return `...`;
  }
};

user.fullName = "Тест"; // Ошибка (у свойства
есть только геттер)
```

Давайте исправим это, добавив сеттер для

`user.fullName`:

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },


};

// set fullName запустится с данным значением
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
  set fullName(value) {
    [this.name, this.surname] = value.split("
");
  }
```

В итоге мы получили «виртуальное» свойство `fullName`. Его можно прочитать и изменить.

Дескрипторы свойств доступа

Дескрипторы свойств-аксессоров отличаются от «обычных» свойств-данных.

Свойства-аксессоры не имеют `value` и `writable`, но взамен предлагают функции `get` и `set`.

То есть, дескриптор аксессора может иметь:

- **`get`** – функция без аргументов, которая сработает при чтении свойства,
- **`set`** – функция, принимающая один аргумент, вызываемая при присвоении свойства,
- **`enumerable`** – то же самое, что и для свойств-данных,

`configurable` – то же самое, что и для свойств-данных.

Например, для создания аксессора `fullName` при помощи `defineProperty` мы можем передать дескриптор с использованием `get` и `set`:

```
let user = {
  name: "John",
  surname: "Smith"
};


});

alert(user.fullName); // John Smith

for(let key in user) alert(key); // name, surname
Object.defineProperty(user, 'fullName', {
  get() {
    return `${this.name} ${this.surname}`;
  },

  set(value) {
    [this.name, this.surname] = value.split("
");
  }
```

Ещё раз заметим, что свойство объекта может быть либо свойством-аксессором (с методами `get/set`), либо свойством-данным (со значением `value`).

При попытке указать и `get`, и `value` в одном дескрипторе будет ошибка:

```
Object.defineProperty({}, 'prop', {
  get() {
    return 1
  },

  value: 2
```

```
});
// Error: Invalid property descriptor.
```

Умные геттеры/сеттеры

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними.

Например, если мы хотим запретить устанавливать короткое имя для `user`, мы можем использовать сеттер `name` для проверки, а само значение хранить в отдельном свойстве `_name`:

```
let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      alert("Имя слишком короткое, должно быть более 4
символов");
      return;
    }
    this._name = value;
  }
};

user.name = "Pete";
alert(user.name); // Pete

user.name = ""; // Имя слишком короткое...
```

Таким образом, само имя хранится в `_name`, доступ к которому производится через геттер и сеттер.

Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью `user._name`, но существует широко известное соглашение о том, что свойства, которые начинаются с символа `"_"`, являются внутренними, и к ним не следует обращаться из-за

пределов объекта.

Использование для совместимости

У аксессоров есть интересная область применения – они позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер.

Например, представим, что мы начали реализовывать объект `user`, используя свойства-данные имя `name` и возраст `age`:

```
function User(name, age) {
  this.name = name;
  this.age = age;
}

let john = new User("John", 25);

alert( john.age ); // 25
```

…Но рано или поздно всё может измениться. Взамен возраста `age` мы можем решить хранить дату рождения `birthday`, потому что так более точно и удобно:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;
}

let john = new User("John", new Date(1992, 6, 1));
```

Что нам делать со старым кодом, который использует свойство `age`?

Мы можем попытаться найти все такие места и изменить их, но это отнимает время и может быть невыполнимо, если код используется другими людьми. И кроме того, `age` – это отличное свойство для `user`, верно?

Давайте его сохраним.

Добавление геттера для `age` решит проблему:

```
function User(name, birthday) {
  this.name = name;
  this.birthday = birthday;


}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday ); // доступен как день рождения
alert( john.age );      // ...так и возраст
  // возраст рассчитывается из текущей даты и
дня рождения
  Object.defineProperty(this, "age", {
    get() {
      let todayYear = new
Date().getFullYear();
      return todayYear -
this.birthday.getFullYear();
    }
  });
```

Теперь старый код тоже работает, и у нас есть отличное дополнительное свойство!

Прототипы, наследование

Прототипное наследование

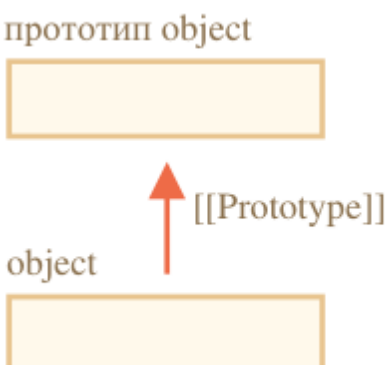
В программировании мы часто хотим взять что-то и расширить.

Например, у нас есть объект `user` со своими свойствами и методами, и мы хотим создать объекты `admin` и `guest` как его слегка изменённые варианты. Мы хотели бы повторно использовать то, что есть у объекта `user`, не копировать/переопределять его методы, а просто создать новый объект на его основе.

Прототипное наследование — это возможность языка, которая помогает в этом.

[[Prototype]]

В JavaScript объекты имеют специальное скрытое свойство `[[Prototype]]` (так оно названо в спецификации), которое либо равно `null`, либо ссылается на другой объект. Этот объект называется «прототип»:



Прототип даёт нам немного «магии». Когда мы хотим прочитать свойство из `object`, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием». Многие интересные возможности языка и техники программирования основываются на нём.

Свойство `[[Prototype]]` является внутренним и скрытым, но есть много способов задать его.

Одним из них является использование `__proto__`, например так:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;
```

Свойство `__proto__` — исторически обусловленный геттер/сеттер для `[[Prototype]]`

Обратите внимание, что `__proto__` — *не то же самое*, что `[[Prototype]]`. Это геттер/сеттер для него.

Он существует по историческим причинам, в современном языке его заменяют функции `Object.getPrototypeOf/Object.setPrototypeOf`, которые также получают/устанавливают прототип. Мы рассмотрим причины этого и сами функции позже.

По спецификации `__proto__` должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его. Далее мы будем в примерах использовать `__proto__`, так как это самый короткий и интуитивно понятный способ установки и чтения прототипа.

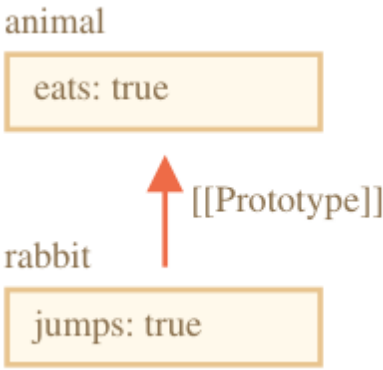
Если мы ищем свойство в `rabbit`, а оно отсутствует, JavaScript автоматически берёт его из `animal`.

Например:

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};


// теперь мы можем найти оба свойства в rabbit:

alert( rabbit.jumps ); // true
rabbit.__proto__ = animal; // (*)
alert( rabbit.eats ); // true (**)
```

Здесь строка `(*)` устанавливает `animal` как прототип для `rabbit`.

Затем, когда `alert` пытается прочитать свойство `rabbit.eats (**)`, его нет в `rabbit`, поэтому JavaScript следует по ссылке `[[Prototype]]` и находит его в `animal` (смотрите снизу вверх):



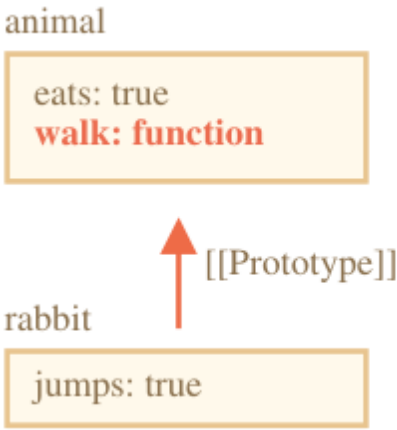
Здесь мы можем сказать, что "`animal` является прототипом `rabbit`" или "`rabbit` прототипно наследует от `animal`".

Так что если у `animal` много полезных свойств и методов, то они автоматически становятся доступными у `rabbit`. Такие свойства называются «унаследованными».

Если у нас есть метод в `animal`, он может быть вызван на `rabbit`:

```
let animal = {
  eats: true,

};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk взят из прототипа
  walk() {
    alert("Animal walk");
  }
rabbit.walk(); // Animal walk
```

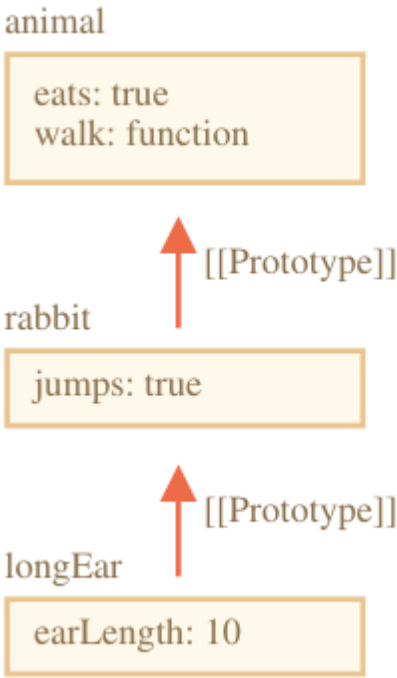
Метод автоматически берётся из прототипа:



Цепочка прототипов может быть длиннее:

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,

};

let longEar = {
  earLength: 10,

};

// walk взят из цепочки прототипов
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (из rabbit)
```

```
__proto__: animal
__proto__: rabbit
```



Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если мы попытаемся назначить `__proto__` по кругу.
2. Значение `__proto__` может быть объектом или `null`. Другие типы игнорируются.

Это вполне очевидно, но всё же: может быть только один `[[Prototype]]`. Объект не может наследоваться от двух других объектов.

Операция записи не использует прототип

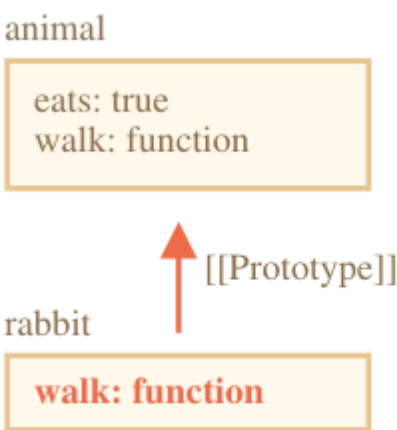
Прототип используется только для чтения свойств.

Операции записи/удаления работают напрямую с объектом.

В приведённом ниже примере мы присваиваем `rabbit` собственный метод `walk`:

```
let animal = {
  eats: true,
  walk() {
    /* этот метод не будет использоваться в rabbit */
  }
};

let rabbit = {
  __proto__: animal
};


rabbit.walk(); // Rabbit! Bounce-bounce!
rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};
```

Теперь вызов `rabbit.walk()` находит метод непосредственно в объекте и выполняет его, не используя прототип:



Свойства-аксессоры – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции.

По этой причине `admin.fullName` работает корректно в приведённом ниже коде:

```
let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// срабатывает сеттер!
admin.fullName = "Alice Cooper"; // (**)
alert(admin.name); // Alice
alert(admin.surname); // Cooper
```

Здесь в строке `(*)` свойство `admin.fullName` имеет геттер в прототипе `user`, поэтому вызывается он. В строке `(**)` свойство также имеет сеттер в прототипе, который и будет вызван.

Значение «this»

В приведённом выше примере может возникнуть интересный вопрос: каково значение `this` внутри `set fullName(value)`? Куда записаны свойства `this.name` и `this.surname`: в `user` или в `admin`?

Ответ прост: прототипы никак не влияют на `this`.

Неважно, где находится метод: в объекте или его

прототипе. При вызове метода `this` — всегда объект перед точкой.

Таким образом, вызов сеттера `admin.fullName=` в качестве `this` использует `admin`, а не `user`.

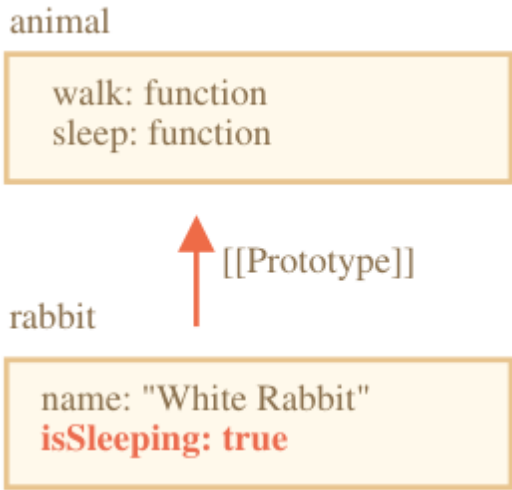
Это на самом деле очень важная деталь, потому что у нас может быть большой объект со множеством методов, от которого можно наследовать. Затем наследующие объекты могут вызывать его методы, но они будут изменять своё состояние, а не состояние объекта-родителя.

Например, здесь `animal` представляет собой «хранилище методов», и `rabbit` использует его.

Вызов `rabbit.sleep()` устанавливает `this.isSleeping` для объекта `rabbit`:

```
// методы animal
let animal = {
  walk() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep() {
    this.isSleeping = true;
  }
};

let rabbit = {
  name: "White Rabbit",
  __proto__: animal
};

// модифицирует rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (нет такого свойства в прототипе)
```

Картинка с результатом:



Если бы у нас были другие объекты, такие как `bird`, `snake` и т.д., унаследованные от `animal`, они также получили бы доступ к методам `animal`. Но `this` при вызове каждого метода будет соответствовать объекту (перед точкой), на котором происходит вызов, а не `animal`. Поэтому, когда мы записываем данные в `this`, они сохраняются в этих объектах.

В результате методы являются общими, а состояние объекта — нет.

Цикл for…in

Цикл `for..in` проходит не только по собственным, но и по унаследованным свойствам объекта.

Например:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};
```

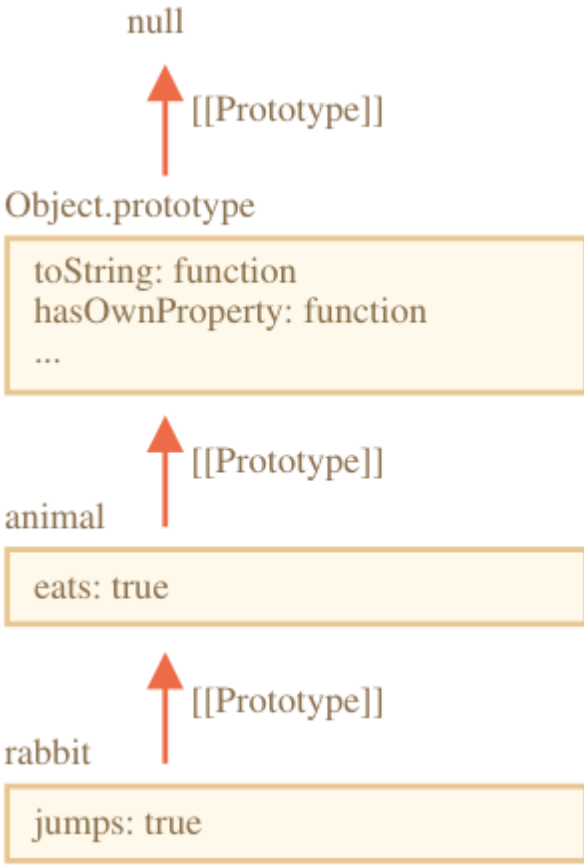
```
// Object.keys возвращает только собственные ключи
alert(Object.keys(rabbit)); // jumps
// for..in проходит и по своим, и по унаследованным ключам
for(let prop in rabbit) alert(prop); // jumps, затем eats
```

Если унаследованные свойства нам не нужны, то мы можем отфильтровать их при помощи встроенного метода obj.hasOwnProperty(key): он возвращает `true`, если у `obj` есть собственное, не унаследованное, свойство с именем `key`.

Пример такой фильтрации:

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

for(let prop in rabbit) {
  let isOwn = rabbit.hasOwnProperty(prop);

  if (isOwn) {
    alert(`Our: ${prop}`); // Our: jumps
  } else {
    alert(`Inherited: ${prop}`); // Inherited: eats
  }
}
```

В этом примере цепочка наследования выглядит так: `rabbit` наследует от `animal`, который наследует от `Object.prototype` (так как `animal` – литеральный объект `{...}`, то это по умолчанию), а затем `null` на самом верху:



Заметим ещё одну деталь. Откуда взялся метод `rabbit.hasOwnProperty`? Мы его явно не определяли. Если посмотреть на цепочку прототипов, то видно, что он берётся из `Object.prototype.hasOwnProperty`. То есть, он унаследован.

…Но почему `hasOwnProperty` не появляется в цикле `for..in` в отличие от `eats` и `jumps`? Он ведь перечисляет все унаследованные свойства.

Ответ простой: оно не перечислимо. То есть, у него внутренний флаг `enumerable` стоит `false`, как и у других свойств `Object.prototype`. Поэтому оно и не появляется в цикле.

Почти все остальные методы получения ключей/значений игнорируют унаследованные свойства

Почти все остальные методы, получающие ключи/значения, такие как `Object.keys`, `Object.values` и другие – игнорируют унаследованные свойства.

Они учитывают только свойства самого объекта, не его прототипа.

Итого

В JavaScript все объекты имеют скрытое свойство `[[Prototype]]`, которое является либо другим объектом, либо `null`.

Мы можем использовать `obj.__proto__` для доступа к нему (исторически обусловленный геттер/сеттер, есть другие способы, которые скоро будут рассмотрены).

Объект, на который ссылается `[[Prototype]]`, называется «прототипом».

Если мы хотим прочитать свойство `obj` или вызвать метод, которого не существует у `obj`, тогда JavaScript попытается найти его в прототипе.

Операции записи/удаления работают непосредственно с объектом, они не используют прототип (если это обычное свойство, а не сеттер).

Если мы вызываем `obj.method()`, а метод при этом взят из прототипа, то `this` всё равно ссылается на `obj`. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.

Цикл `for..in` перебирает как свои, так и унаследованные свойства. Остальные методы получения ключей/значений работают только с собственными свойствами объекта.

Задачи

Работа с прототипами

важность: 5

В приведённом ниже коде создаются и изменяются два объекта.

Какие значения показываются в процессе выполнения кода?

```
let animal = {
  jumps: null
};
let rabbit = {
  __proto__: animal,
  jumps: true
};

alert( rabbit.jumps ); // ? (1)

delete rabbit.jumps;

alert( rabbit.jumps ); // ? (2)

delete animal.jumps;

alert( rabbit.jumps ); // ? (3)
```

Должно быть 3 ответа.

К решению

Алгоритм поиска

важность: 5

Задача состоит из двух частей.

У нас есть объекты:

```
let head = {
  glasses: 1
};

let table = {
  pen: 3
};

let bed = {
  sheet: 1,
  pillow: 2
};

let pockets = {
  money: 2000
};
```

1. С помощью свойства `__proto__` задайте прототипы так, чтобы поиск любого свойства выполнялся по следующему пути: `pockets` → `bed` → `table` → `head`. Например, `pockets.pen` должно возвращать значение `3` (найденное в `table`), а `bed.glasses` – значение `1` (найденное в `head`).
2. Ответьте на вопрос: как быстрее получить значение `glasses` – через `pockets.glasses` или через `head.glasses`? При необходимости составьте цепочки поиска и сравните их.

К решению

Куда будет произведена запись?

важность: 5

Объект `rabbit` наследует от объекта `animal`.

Какой объект получит свойство `full` при вызове

`rabbit.eat()`: `animal` или `rabbit`?

```
let animal = {
  eat() {
    this.full = true;
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.eat();
```

К решению

Почему наедаются оба хомяка?

важность: 5

У нас есть два хомяка: шустрый (`speedy`) и ленивый (`lazy`); оба наследуют от общего объекта `hamster`.

Когда мы кормим одного хомяка, второй тоже наедается. Почему? Как это исправить?

```
let hamster = {
  stomach: [],

  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// Этот хомяк нашёл еду
speedy.eat("apple");
alert( speedy.stomach ); // apple
```

```
// У этого хомяка тоже есть еда. Почему? Исправьте
alert( lazy.stomach ); // apple
```

К решению

F.prototype

Как мы помним, новые объекты могут быть созданы с помощью функции-конструктора `new F()`.

Если в `F.prototype` содержится объект, оператор `new` устанавливает его в качестве `[[Prototype]]` для нового объекта.

> **На заметку:**
>
> JavaScript использовал прототипное наследование с момента своего появления. Это одна из основных особенностей языка.
>
> Но раньше, в старые времена, прямого доступа к прототипу объекта не было. Надёжно работало только свойство `"prototype"` функции-конструктора, описанное в этой главе. Поэтому оно используется во многих скриптах.

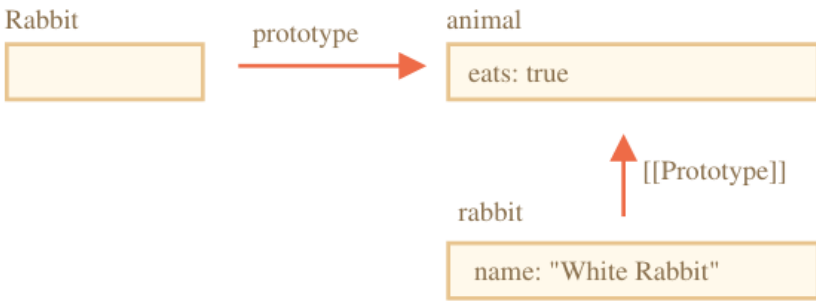
Обратите внимание, что `F.prototype` означает обычное свойство с именем `"prototype"` для `F`. Это ещё не «прототип объекта», а обычное свойство `F` с таким именем.

Приведём пример:

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}


let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__
== animal

alert( rabbit.eats ); // true
Rabbit.prototype = animal;
```

Установка `Rabbit.prototype = animal` буквально говорит интерпретатору следующее: "При создании объекта через `new Rabbit()` запиши ему `animal` в `[[Prototype]]`".

Результат будет выглядеть так:



На изображении: `"prototype"` – горизонтальная стрелка, обозначающая обычное свойство для `"F"`, а `[[Prototype]]` – вертикальная, обозначающая наследование `rabbit` от `animal`.

`F.prototype` используется только в момент вызова `new F()`

`F.prototype` используется только при вызове `new F()` и присваивается в качестве свойства `[[Prototype]]` нового объекта. После этого `F.prototype` и новый объект ничего не связывает. Следует понимать это как «единоразовый подарок» объекту.

После создания `F.prototype` может измениться, и новые объекты, созданные с помощью `new F()`, будут иметь другой объект в качестве `[[Prototype]]`, но уже существующие объекты сохранят старый.

F.prototype по умолчанию, свойство constructor

У каждой функции по умолчанию уже есть свойство `"prototype"`.

По умолчанию `"prototype"` – объект с единственным свойством `constructor`, которое ссылается на функцию-конструктор.

Вот такой:

```
function Rabbit() {}

/* прототип по умолчанию
Rabbit.prototype = { constructor: Rabbit };
*/
```

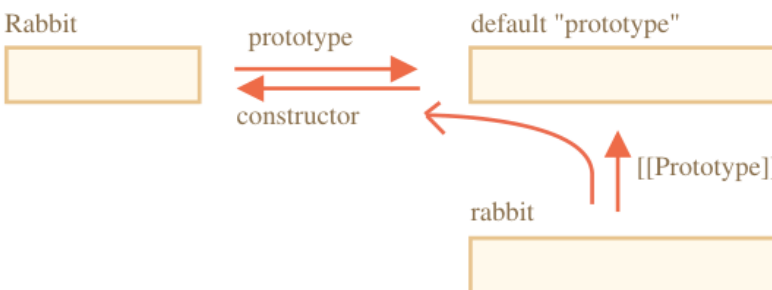
Проверим это:

```
function Rabbit() {}
```

```
// по умолчанию:
// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Соответственно, если мы ничего не меняем, то свойство `constructor` будет доступно всем кроликам через `[[Prototype]]`:

```
function Rabbit() {}
// по умолчанию:
// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // наследует от {constructor:
Rabbit}

alert(rabbit.constructor == Rabbit); // true (свойство
получено из прототипа)
```



Мы можем использовать свойство `constructor` существующего объекта для создания нового.

Пример:

```
function Rabbit(name) {
  this.name = name;
  alert(name);
}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black
Rabbit");
```

Это удобно, когда у нас есть объект, но мы не знаем, какой конструктор использовался для его создания (например, он мог быть взят из сторонней библиотеки), а нам необходимо создать ещё один такой объект.

Но, пожалуй, самое важное о свойстве `"constructor"` это то, что…

…JavaScript сам по себе не гарантирует правильное значение свойства `"constructor"`.

Да, оно является свойством по умолчанию в `"prototype"` у функций, но что случится с ним позже – зависит только от нас.

В частности, если мы заменим прототип по умолчанию на другой объект, то свойства `"constructor"` в нём не будет.

Например:

```
function Rabbit() {}
Rabbit.prototype = {
  jumps: true
};

let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

Таким образом, чтобы сохранить верное свойство `"constructor"`, мы должны добавлять/удалять/изменять свойства у прототипа по умолчанию вместо того, чтобы перезаписывать его целиком:

```
function Rabbit() {}

// Не перезаписываем Rabbit.prototype полностью,
// а добавляем к нему свойство
Rabbit.prototype.jumps = true
// Прототип по умолчанию сохраняется, и мы всё ещё имеем
доступ к Rabbit.prototype.constructor
```

Или мы можем заново создать свойство `constructor`:

```
Rabbit.prototype = {
  jumps: true,

};

// теперь свойство constructor снова корректное, так как мы добавили его
  constructor: Rabbit
```

Итого

В этой главе мы кратко описали способ задания `[[Prototype]]` для объектов, создаваемых с помощью функции-конструктора. Позже мы рассмотрим, как можно использовать эту возможность.

Всё достаточно просто. Выделим основные моменты:

- Свойство `F.prototype` (не путать с `[[Prototype]]`) устанавливает `[[Prototype]]` для новых объектов при вызове `new F()`.
- Значение `F.prototype` должно быть либо объектом, либо `null`. Другие значения не будут работать.
- Свойство `"prototype"` является особым, только когда оно назначено функции-конструктору, которая вызывается оператором `new`.

В обычных объектах `prototype` не является чем-то особенным:

```
let user = {
  name: "John",
  prototype: "Bla-bla" // никакой магии нет - обычное свойство
};
```

По умолчанию все функции имеют `F.prototype = {`

`constructor: F }`, поэтому мы можем получить конструктор объекта через свойство `"constructor"`.

Задачи

Изменяем "prototype"

важность: 5

В коде ниже мы создаём нового кролика `new Rabbit`, а потом пытаемся изменить его прототип.

Сначала у нас есть такой код:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();

alert( rabbit.eats ); // true
```

1. Добавим одну строчку (выделенную в коде ниже). Что вызов `alert` покажет нам сейчас?

   ```
   function Rabbit() {}
   Rabbit.prototype = {
     eats: true
   };

   let rabbit = new Rabbit();



   alert( rabbit.eats ); // ?
   Rabbit.prototype = {};
   ```

2. …А если код такой (заменили одну строчку)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();



alert( rabbit.eats ); // ?
Rabbit.prototype.eats = false;
```

3. Или такой (заменили одну строчку)?

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();



alert( rabbit.eats ); // ?
delete rabbit.eats;
```

4. Или, наконец, такой:

```
function Rabbit() {}
Rabbit.prototype = {
  eats: true
};

let rabbit = new Rabbit();



alert( rabbit.eats ); // ?
delete Rabbit.prototype.eats;
```

К решению

Создайте новый объект с помощью уже существующего

важность: 5

Представьте, что у нас имеется некий объект `obj`, созданный функцией-конструктором – мы не знаем какой именно, но хотелось бы создать ещё один объект такого же типа.

Можем ли мы сделать так?

```
let obj2 = new obj.constructor();
```

Приведите пример функции-конструктора для объекта `obj`, с которой такой вызов корректно сработает. И пример функции-конструктора, с которой такой код поведёт себя неправильно.

К решению

Встроенные прототипы

Свойство `"prototype"` широко используется внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

Сначала мы рассмотрим детали, а затем используем `"prototype"` для добавления встроенным объектам новой функциональности.

Object.prototype

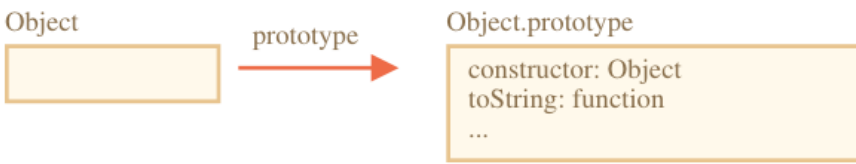
Давайте выведем пустой объект:

```
let obj = {};
alert( obj ); // "[object Object]" ?
```

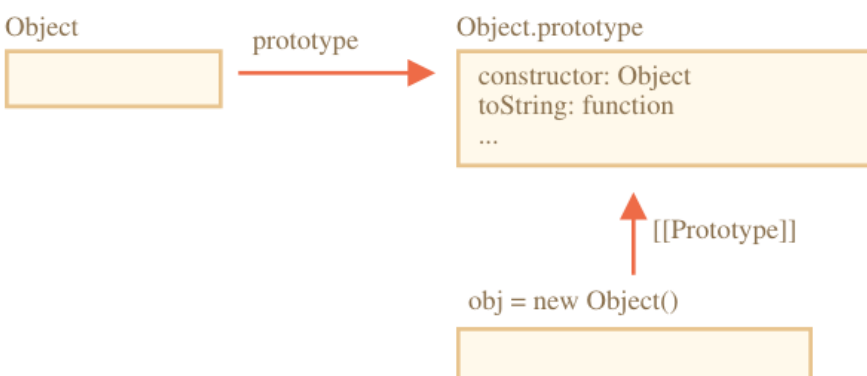
Где код, который генерирует строку `"[object Object]"`? Это встроенный метод `toString`, но где он? `obj` ведь пуст!

...Но краткая нотация `obj = {}` – это то же самое, что и `obj = new Object()`, где `Object` – встроенная функция-конструктор для объектов с собственным свойством `prototype`, которое ссылается на огромный объект с методом `toString` и другими.

Вот что происходит:



Когда вызывается `new Object()` (или создаётся объект с помощью литерала `{...}`), свойство `[[Prototype]]` этого объекта устанавливается на `Object.prototype` по правилам, которые мы обсуждали в предыдущей главе:



Таким образом, когда вызывается `obj.toString()`, метод берётся из `Object.prototype`.

Мы можем проверить это так:

```
let obj = {};

alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString ==
Object.prototype.toString
```

Обратите внимание, что по цепочке прототипов выше `Object.prototype` больше нет свойства `[[Prototype]]`:

```
alert(Object.prototype.__proto__); // null
```

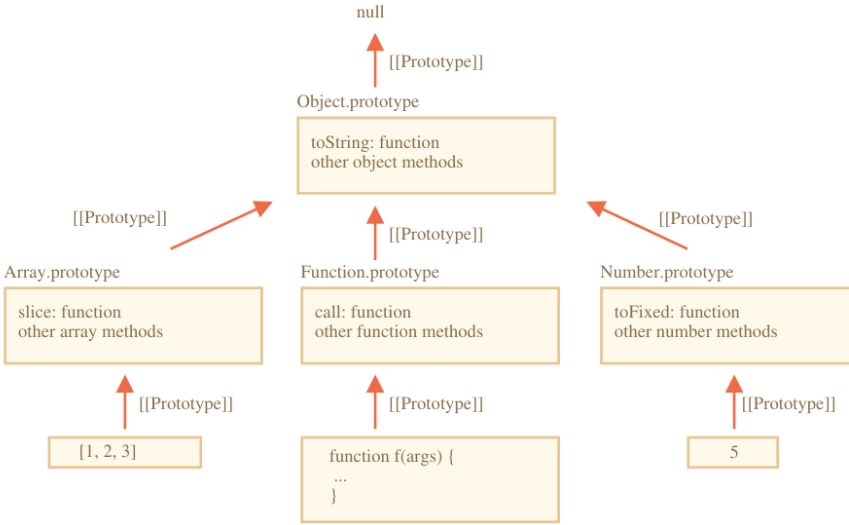
Другие встроенные прототипы

Другие встроенные объекты, такие как `Array`, `Date`, `Function` и другие, также хранят свои методы в прототипах.

Например, при создании массива `[1, 2, 3]` внутренне используется конструктор массива `Array`. Поэтому прототипом массива становится `Array.prototype`, предоставляя ему свои методы. Это позволяет эффективно использовать память.

Согласно спецификации, наверху иерархии встроенных прототипов находится `Object.prototype`. Поэтому иногда говорят, что «всё наследует от объектов».

Вот более полная картина (для трёх встроенных объектов):



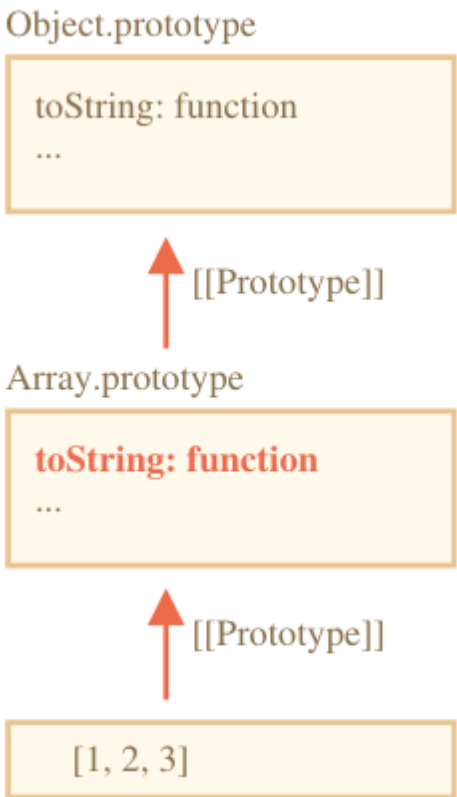
Давайте проверим прототипы:

```
let arr = [1, 2, 3];

// наследует ли от Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// затем наследует ли от Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// и null на вершине иерархии
alert( arr.__proto__.__proto__.__proto__ ); // null
```

Некоторые методы в прототипах могут пересекаться, например, у `Array.prototype` есть свой метод `toString`, который выводит элементы массива через запятую:

```
let arr = [1, 2, 3]
alert(arr); // 1,2,3 <-- результат Array.prototype.toString
```

Как мы видели ранее, у `Object.prototype` есть свой метод `toString`, но так как `Array.prototype` ближе в цепочке прототипов, то берётся именно вариант для массивов:



В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования (возможно, потребуется использовать `console.dir` для встроенных объектов):

```
> console.dir([1,2,3])
  ▼ Array[3] i
      0: 1
      1: 2
      2: 3
      length: 3
    ▼ __proto__:=Array.prototype
      ▶ concat: function concat() { [native code] }
      ▶ ...
      ▶ unshift: function unshift() { [native code] }
      ▼ __proto__:=Object.prototype
        ▶ ...
        ▶ constructor: function Object() { [native code] }
        ▶ hasOwnProperty: function hasOwnProperty() { [native code] }
        ▶ isPrototypeOf: function isPrototypeOf() { [native code] }
        ▶ ...
```

Другие встроенные объекты устроены аналогично. Даже функции – они объекты встроенного конструктора `Function`, и все их методы (`call`/`apply` и другие) берутся из `Function.prototype`. Также у функций есть свой метод `toString`.

```
function f() {}

alert(f.__proto__ == Function.prototype); // true
alert(f.__proto__.__proto__ == Object.prototype); // true,
наследует от Object
```

Примитивы

Самое сложное происходит со строками, числами и булевыми значениями.

Как мы помним, они не объекты. Но если мы попытаемся получить доступ к их свойствам, то тогда будет создан временный объект-обёртка с использованием встроенных конструкторов `String`, `Number` и `Boolean`, который предоставит методы и после этого исчезнет.

Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как `String.prototype`, `Number.prototype` и `Boolean.prototype`.

Значения `null` и `undefined` не имеют объектов-обёрток

Специальные значения `null` и `undefined` стоят особняком. У них нет объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

Изменение встроенных прототипов

Встроенные прототипы можно изменять. Например, если добавить метод к `String.prototype`, метод становится доступен для всех строк:

```
String.prototype.show = function() {
  alert(this);
};

"BOOM!".show(); // BOOM!
```

В течение процесса разработки у нас могут возникнуть идеи о новых встроенных методах, которые нам хотелось бы иметь, и искушение добавить их во встроенные прототипы. Это плохая идея.

> **Важно:**
>
> Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод `String.prototype.show`, то одна из них перепишет метод другой.
>
> Так что, в общем, изменение встроенных прототипов считается плохой идеей.

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов.

Полифил – это термин, который означает эмуляцию метода, который существует в спецификации JavaScript, но ещё не поддерживается текущим движком JavaScript.

Тогда мы можем реализовать его сами и добавить во встроенный прототип.

Например:

```
if (!String.prototype.repeat) { // Если такого метода нет
  // добавляем его в прототип

  String.prototype.repeat = function(n) {
    // повторить строку n раз

    // на самом деле код должен быть немного более сложным
    // (полный алгоритм можно найти в спецификации)
    // но даже неполный полифил зачастую достаточно хорош для
использования
    return new Array(n + 1).join(this);
  };
}

alert( "La".repeat(3) ); // LaLaLa
```

Заимствование у прототипов

В главе Декораторы и переадресация вызова, call/apply мы говорили о заимствовании методов.

Это когда мы берём метод из одного объекта и копируем его в другой.

Некоторые методы встроенных прототипов часто одалживают.

Например, если мы создаём объект, похожий на массив (псевдомассив), мы можем скопировать некоторые методы из `Array` в этот объект.

Пример:

```
let obj = {
  0: "Hello",
  1: "world!",
  length: 2,
};
```

```
alert( obj.join(',') ); // Hello,world!
obj.join = Array.prototype.join;
```

Это работает, потому что для внутреннего алгоритма встроенного метода `join` важны только корректность индексов и свойство `length`, он не проверяет, является ли объект на самом деле массивом. И многие встроенные методы работают так же.

Альтернативная возможность – мы можем унаследовать от массива, установив `obj.__proto__` как `Array.prototype`, таким образом все методы `Array` станут автоматически доступны в `obj`.

Но это будет невозможно, если `obj` уже наследует от другого объекта. Помните, мы можем наследовать только от одного объекта одновременно.

Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

Итого

Все встроенные объекты следуют одному шаблону:

- Методы хранятся в прототипах (`Array.prototype`, `Object.prototype`, `Date.prototype` и т.д.).
- Сами объекты хранят только данные (элементы массивов, свойства объектов, даты).

Примитивы также хранят свои методы в прототипах объектов-обёрток: `Number.prototype`, `String.prototype`, `Boolean.prototype`. Только у значений `undefined` и `null` нет объектов-обёрток.

Встроенные прототипы могут быть изменены или дополнены новыми методами. Но не рекомендуется менять их. Единственная допустимая причина – это добавление нового метода из стандарта, который ещё

не поддерживается движком JavaScript.

Задачи

Добавить функциям метод "f.defer(ms)"

важность: 5

Добавьте всем функциям в прототип метод `defer(ms)`, который вызывает функции через `ms` миллисекунд.

После этого должен работать такой код:

```
function f() {
  alert("Hello!");
}

f.defer(1000); // выведет "Hello!" через 1 секунду
```

К решению

Добавьте функциям декорирующий метод "defer()"

важность: 4

Добавьте всем функциям в прототип метод `defer(ms)`, который возвращает обёртку, откладывающую вызов функции на `ms` миллисекунд.

Например, должно работать так:

```
function f(a, b) {
  alert( a + b );
}
```

```
f.defer(1000)(1, 2); // выведет 3 через 1 секунду.
```

Пожалуйста, заметьте, что аргументы должны корректно передаваться оригинальной функции.

К решению

Методы прототипов, объекты без свойства __proto__

В первой главе этого раздела мы упоминали, что существуют современные методы работы с прототипами.

Свойство `__proto__` считается устаревшим, и по стандарту оно должно поддерживаться только браузерами.

Современные же методы это:

- Object.create(proto, [descriptors]) – создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto`, и необязательными дескрипторами свойств `descriptors`.
- Object.getPrototypeOf(obj) – возвращает свойство `[[Prototype]]` объекта `obj`.
- Object.setPrototypeOf(obj, proto) – устанавливает свойство `[[Prototype]]` объекта `obj` как `proto`.

Эти методы нужно использовать вместо `__proto__`.

Например:

```
let animal = {
  eats: true
```

```
};

// создаём новый объект с прототипом animal

alert(rabbit.eats); // true
```

```
let rabbit = Object.create(animal);
alert(Object.getPrototypeOf(rabbit) ===
animal); // получаем прототип объекта rabbit
Object.setPrototypeOf(rabbit, {}); // заменяем
прототип объекта rabbit на {}
```

У `Object.create` есть необязательный второй аргумент: дескрипторы свойств. Мы можем добавить дополнительное свойство новому объекту таким образом:

```
let animal = {
  eats: true
};

let rabbit = Object.create(animal, {
  jumps: {
    value: true
  }
});

alert(rabbit.jumps); // true
```

Формат задания дескрипторов описан в главе Флаги и дескрипторы свойств.

Мы также можем использовать `Object.create` для «продвинутого» клонирования объекта, более мощного, чем копирование свойств в цикле `for..in`:

```
// клон obj с тем же прототипом (с поверхностным копированием
свойств)
let clone = Object.create(Object.getPrototypeOf(obj),
Object.getOwnPropertyDescriptors(obj));
```

Такой вызов создаёт точную копию объекта `obj`, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством `[[Prototype]]`.

Краткая история

Если пересчитать все способы управления прототипом, то их будет много! И многие из них делают одно и то же!

Почему так?

В силу исторических причин.

> Свойство `"prototype"` функции-конструктора существует с совсем давних времён.
>
> Позднее, в 2012 году, в стандарте появился метод `Object.create`. Это давало возможность создавать объекты с указанным прототипом, но не позволяло устанавливать/получать его. Тогда браузеры реализовали нестандартный аксессор `__proto__`, который позволил устанавливать/получать прототип в любое время.
>
> Позднее, в 2015 году, в стандарт были добавлены `Object.setPrototypeOf` и `Object.getPrototypeOf`, заменяющие собой аксессор `__proto__`, который упоминается в Приложении Б стандарта, которое не обязательно к поддержке в небраузерных окружениях. При этом де-факто `__proto__` всё ещё поддерживается везде.

В итоге сейчас у нас есть все эти способы для работы с прототипом.

Почему же `__proto__` был заменён на функции `getPrototypeOf/setPrototypeOf`? Читайте далее, чтобы узнать ответ.

Не меняйте `[[Prototype]]` существующих объектов, если важна скорость

Технически мы можем установить/получить `[[Prototype]]` в любое время. Но обычно мы устанавливаем прототип только раз во время создания объекта, а после не меняем: `rabbit` наследует от `animal`, и это не изменится.

И JavaScript движки хорошо оптимизированы для этого. Изменение прототипа «на лету» с помощью `Object.setPrototypeOf` или `obj.__proto__=` – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого кроме тех случаев, когда вы знаете, что делаете, или же когда скорость JavaScript для вас не имеет никакого значения.

"Простейший" объект

Как мы знаем, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение.

…Но если мы попробуем хранить *созданные пользователями* ключи (например, словари с пользовательским вводом), мы можем заметить интересный сбой: все ключи работают как ожидается, за исключением `"__proto__"`.

Посмотрите на пример:

```
let obj = {};

let key = prompt("What's the key?", "__proto__");
```

```
obj[key] = "some value";

alert(obj[key]); // [object Object], не "some value"!
```

Если пользователь введёт `__proto__`, присвоение проигнорируется!

И это не должно удивлять нас. Свойство `__proto__` особенное: оно должно быть либо объектом, либо `null`, а строка не может стать прототипом.

Но мы не *намеревались* реализовывать такое поведение, не так ли? Мы хотим хранить пары ключ/значение, и ключ с именем `"__proto__"` не был сохранён надлежащим образом. Так что это ошибка!

Конкретно в этом примере последствия не так ужасны, но если мы присваиваем объектные значения, то прототип и в самом деле может быть изменён. В результате дальнейшее выполнение пойдёт совершенно непредсказуемым образом.

Что хуже всего – разработчики не задумываются о такой возможности совсем. Это делает такие ошибки сложным для отлавливания или даже превращает их в уязвимости, особенно когда JavaScript используется на сервере.

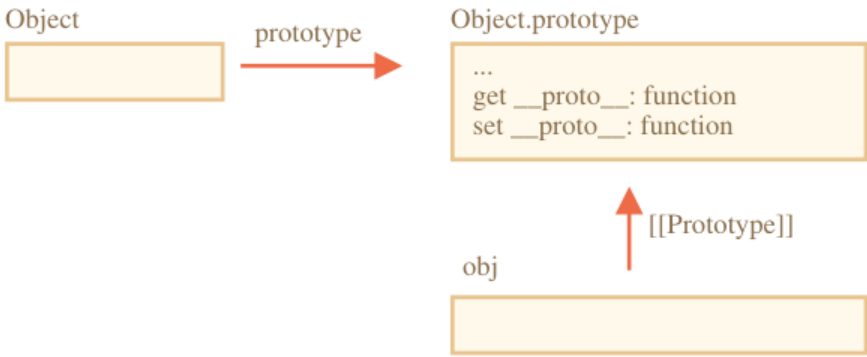
Неожиданные вещи могут случаться также при присвоении свойства `toString`, которое по умолчанию функция, и других свойств, которые тоже на самом деле являются встроенными методами.

Как же избежать проблемы?

Во-первых, мы можем переключиться на использование коллекции `Map`, и тогда всё будет в порядке.

Но и `Object` может также хорошо подойти, потому что создатели языка уже давно продумали решение проблемы.

Свойство `__proto__` – не обычное, а аксессор, заданный в `Object.prototype`:



Так что при чтении или установке `obj.__proto__` вызывается соответствующий геттер/сеттер из прототипа `obj`, и именно он устанавливает/получает свойство `[[Prototype]]`.

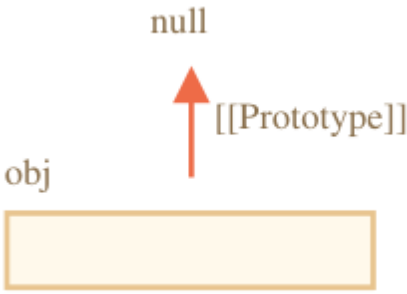
Как было сказано в начале этой секции учебника, `__proto__` – это способ доступа к свойству `[[Prototype]]`, это не само свойство `[[Prototype]]`.

Теперь, если мы хотим использовать объект как ассоциативный массив, мы можем сделать это с помощью небольшого трюка:

```
let key = prompt("What's the key?", "__proto__");
obj[key] = "some value";

alert(obj[key]); // "some value"
```

```
let obj = Object.create(null);
```

`Object.create(null)` создаёт пустой объект без прототипа (`[[Prototype]]` будет `null`):



Таким образом не будет унаследованного геттера/сеттера для `__proto__`. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно.

Мы можем назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще, чем обычные объекты `{...}`.

Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как `toString`:

```
alert(obj); // Error (no toString)
let obj = Object.create(null);
```

…Но обычно это нормально для ассоциативных массивов.

Обратите внимание, что большая часть методов, связанных с объектами, имеют вид `Object.something(...)`. К примеру, `Object.keys(obj)`. Подобные методы не находятся в прототипе, так что они продолжат работать для таких объектов:

```
let chineseDictionary = Object.create(null);
chineseDictionary.hello = "你好";
chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye
```

Итого

Современные способы установки и прямого доступа к прототипу это:

- Object.create(proto[, descriptors]) – создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto` (может быть `null`), и необязательными дескрипторами свойств.
- Object.getPrototypeOf(obj) – возвращает свойство `[[Prototype]]` объекта `obj` (то же самое, что и геттер `__proto__`).
- Object.setPrototypeOf(obj, proto) – устанавливает свойство `[[Prototype]]` объекта `obj` как `proto` (то же самое, что и сеттер `__proto__`).

Встроенный геттер/сеттер `__proto__` не безопасен, если мы хотим использовать *созданные пользователями* ключи в объекте. Как минимум потому, что пользователь может ввести `"__proto__"` как ключ, от чего может возникнуть ошибка. Если повезёт – последствия будут лёгкими, но, вообще говоря, они непредсказуемы.

Так что мы можем использовать либо `Object.create(null)` для создания «простейшего» объекта, либо использовать коллекцию `Map`.

Кроме этого, `Object.create` даёт нам лёгкий способ создать поверхностную копию объекта со всеми дескрипторами:

```
let clone = Object.create(Object.getPrototypeOf(obj),
Object.getOwnPropertyDescriptors(obj));
```

Мы также ясно увидели, что `__proto__` – это геттер/сеттер для свойства `[[Prototype]]`, и находится он в `Object.prototype`, как и другие методы.

Мы можем создавать объекты без прототипов с помощью `Object.create(null)`. Такие объекты можно использовать как «чистые словари», у них нет проблем с использованием строки `"__proto__"` в качестве ключа.

Ещё методы:

- Object.keys(obj) / Object.values(obj) / Object.entries(obj) – возвращают массив всех перечисляемых собственных строковых ключей/значений/пар ключ-значение.
- Object.getOwnPropertySymbols(obj) – возвращает массив всех собственных символьных ключей.
- Object.getOwnPropertyNames(obj) – возвращает массив всех собственных строковых ключей.
- Reflect.ownKeys(obj) – возвращает массив всех собственных ключей.
- obj.hasOwnProperty(key): возвращает `true`, если у `obj` есть собственное (не унаследованное) свойство с именем `key`.

Все методы, которые возвращают свойства объектов (такие как `Object.keys` и другие), возвращают «собственные» свойства. Если мы хотим получить и унаследованные, можно воспользоваться циклом `for..in`.

Задачи

Добавьте toString в словарь

важность: 5

Имеется объект `dictionary`, созданный с помощью `Object.create(null)` для хранения любых пар `ключ/значение`.

Добавьте ему метод `dictionary.toString()`, который должен возвращать список ключей, разделённых запятой. Ваш `toString` не должен выводиться при итерации объекта с помощью цикла `for..in`.

Вот так это должно работать:

```
let dictionary = Object.create(null);



// добавляем немного данных
dictionary.apple = "Apple";
dictionary.__proto__ = "test"; // здесь __proto__ -- это обычный ключ

// только apple и __proto__ выведены в цикле
for(let key in dictionary) {
  alert(key); // "apple", затем "__proto__"
}

// ваш метод toString в действии
alert(dictionary); // "apple,__proto__"
// ваш код, который добавляет метод dictionary.toString
```

К решению

Разница между вызовами

важность: 5

Давайте создадим новый объект `rabbit`:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Rabbit");
```

Все эти вызовы делают одно и тоже или нет?

```
rabbit.sayHi();
Rabbit.prototype.sayHi();
Object.getPrototypeOf(rabbit).sayHi();
rabbit.__proto__.sayHi();
```

К решению

Классы

Класс: базовый синтаксис

> *В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).*
> Википедия

На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё.

Как мы уже знаем из главы Конструкторы, создание объектов через "new", с этим может помочь `new function`.

Но в современном JavaScript есть и более продвинутая конструкция «class», которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

Синтаксис «class»

Базовый синтаксис выглядит так:

```
class MyClass {
  // методы класса
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }
  ...
}
```

Затем используйте вызов `new MyClass()` для создания нового объекта со всеми перечисленными методами.

При этом автоматически вызывается метод `constructor()`, в нём мы можем инициализировать объект.

Например:

```
class User {

  constructor(name) {
    this.name = name;
  }

  sayHi() {
    alert(this.name);
  }

}

// Использование:
let user = new User("Иван");
user.sayHi();
```

Когда вызывается `new User("Иван")`:

1. Создаётся новый объект.
2. `constructor` запускается с заданным аргументом и сохраняет его в `this.name`.

…Затем можно вызывать на объекте методы, такие как `user.sayHi()`.

Методы в классе не разделяются запятой

Частая ошибка начинающих разработчиков – ставить запятую между методами класса, что приводит к синтаксической ошибке.

Синтаксис классов отличается от литералов объектов, не путайте их. Внутри классов запятые не требуются.

Что такое класс?

Итак, что же такое `class`? Это не полностью новая языковая сущность, как может показаться на первый взгляд.

Давайте развеем всю магию и посмотрим, что такое класс на самом деле. Это поможет в понимании многих сложных аспектов.

В JavaScript класс – это разновидность функции.

Взгляните:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// доказательство: User - это функция
alert(typeof User); // function
```

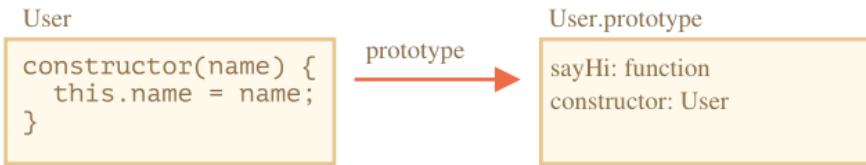
Вот что на самом деле делает конструкция `class User {...}`:

1. Создаёт функцию с именем `User`, которая становится результатом объявления класса. Код функции берётся из метода `constructor` (она будет пустой, если

такого метода нет).
2. Сохраняет все методы, такие как `sayHi`, в `User.prototype`.

При вызове метода объекта `new User` он будет взят из прототипа, как описано в главе F.prototype. Таким образом, объекты `new User` имеют доступ к методам класса.

На картинке показан результат объявления `class User`:



Можно проверить вышесказанное и при помощи кода:

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}

// класс – это функция
alert(typeof User); // function

// ...или, если точнее, это метод constructor
alert(User === User.prototype.constructor); // true

// Методы находятся в User.prototype, например:
alert(User.prototype.sayHi); // alert(this.name);

// в прототипе ровно 2 метода
alert(Object.getOwnPropertyNames(User.prototype)); //
constructor, sayHi
```

Не просто синтаксический сахар

Иногда говорят, что `class` – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового),

потому что мы можем сделать всё то же самое без конструкции `class`:

```
// перепишем класс User на чистых функциях

// 1. Создаём функцию constructor
function User(name) {
  this.name = name;
}
// каждый прототип функции имеет свойство constructor по
умолчанию,
// поэтому нам нет необходимости его создавать

// 2. Добавляем метод в прототип
User.prototype.sayHi = function() {
  alert(this.name);
};

// Использование:
let user = new User("Иван");
user.sayHi();
```

Результат этого кода очень похож. Поэтому, действительно, есть причины, по которым `class` можно считать синтаксическим сахаром для определения конструктора вместе с методами прототипа.

Однако есть важные отличия:

1. Во-первых, функция, созданная с помощью `class`, помечена специальным внутренним свойством `[[FunctionKind]]:"classConstructor"`. Поэтому это не совсем то же самое, что создавать её вручную.

   В отличие от обычных функций, конструктор класса не может быть вызван без `new`:

   ```
   class User {
     constructor() {}
   }

   alert(typeof User); // function
   User(); // Error: Class constructor User cannot be invoked
   without 'new'
   ```

Кроме того, строковое представление конструктора класса в большинстве движков JavaScript начинается с «class …»

```
class User {
  constructor() {}
}

alert(User); // class User { ... }
```

2. Методы класса являются неперечислимыми. Определение класса устанавливает флаг `enumerable` в`false` для всех методов в `"prototype"`.

   И это хорошо, так как если мы проходимся циклом `for..in` по объекту, то обычно мы не хотим при этом получать методы класса.

3. Классы всегда используют `use strict`. Весь код внутри класса автоматически находится в строгом режиме.

Также в дополнение к основной, описанной выше, функциональности, синтаксис `class` даёт ряд других интересных возможностей, с которыми мы познакомимся чуть позже.

Class Expression

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.

Пример Class Expression (по аналогии с Function Expression):

```
let User = class {
  sayHi() {
    alert("Привет");
  }
```

```
};
```

Аналогично Named Function Expression, Class Expression может иметь имя.

Если у Class Expression есть имя, то оно видно только внутри класса:

```
// "Named Class Expression"
// (в спецификации нет такого термина, но происходящее похоже на Named Function Expression)
let User = class  {
  sayHi() {
    alert(MyClass); // имя MyClass видно только внутри класса
  }
};

new User().sayHi(); // работает, выводит определение MyClass

alert(MyClass); // ошибка, имя MyClass не видно за пределами класса
```

```
MyClass
```

Мы даже можем динамически создавать классы «по запросу»:

```
function makeClass(phrase) {
  // объявляем класс и возвращаем его
  return class {
    sayHi() {
      alert(phrase);
    };
  };
}

// Создаём новый класс
let User = makeClass("Привет");

new User().sayHi(); // Привет
```

Геттеры/сеттеры, другие сокращения

Как и в литеральных объектах, в классах можно объявлять

вычисляемые свойства, геттеры/сеттеры и т.д.

Вот пример `user.name`, реализованного с использованием `get/set`:

```
class User {

  constructor(name) {
    // вызывает сеттер
    this.name = name;
  }


    return this._name;
  }


    if (value.length < 4) {
      alert("Имя слишком короткое.");
      return;
    }
    this._name = value;
  }

}

let user = new User("Иван");
alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.
```

```
  get name() {
  set name(value) {
```

При объявлении класса геттеры/сеттеры создаются на `User.prototype`, вот так:

```
Object.defineProperties(User.prototype, {
  name: {
    get() {
      return this._name
    },
    set(name) {
      // ...
    }
  }
});
```

Пример с вычисляемым свойством в скобках `[...]`:

```
class User {

  ['say' + 'Hi']() {
    alert("Привет");
  }

}

new User().sayHi();
```

Свойства классов

> **Старым браузерам может понадобиться полифил**
>
> Свойства классов добавлены в язык недавно.

В приведённом выше примере у класса `User` были только методы. Давайте добавим свойство:

```
class User {


  sayHi() {
    alert(`Привет, ${this.name}!`);
  }
}

new User().sayHi();
  name = "Аноним";
```

Свойство `name` не устанавливается в `User.prototype`. Вместо этого оно создаётся оператором `new` перед запуском конструктора, это именно свойство объекта.

Итого

Базовый синтаксис для классов выглядит так:

```
class MyClass {
  prop = value; // свойство
  constructor(...) { // конструктор
    // ...
  }
  method(...) {} // метод
  get something(...) {} // геттер
  set something(...) {} // сеттер
  [Symbol.iterator]() {} // метод с вычисляемым именем (здесь
- символом)
  // ...
}
```

`MyClass` технически является функцией (той, которую мы определяем как `constructor`), в то время как методы, геттеры и сеттеры записываются в `MyClass.prototype`.

В следующих главах мы узнаем больше о классах, включая наследование и другие возможности.

Задачи

Перепишите класс

важность: 5

Класс `Clock` написан в функциональном стиле. Перепишите его, используя современный синтаксис классов.

P.S. Часики тикают в консоли. Откройте её, чтобы посмотреть.

Открыть песочницу для задачи.

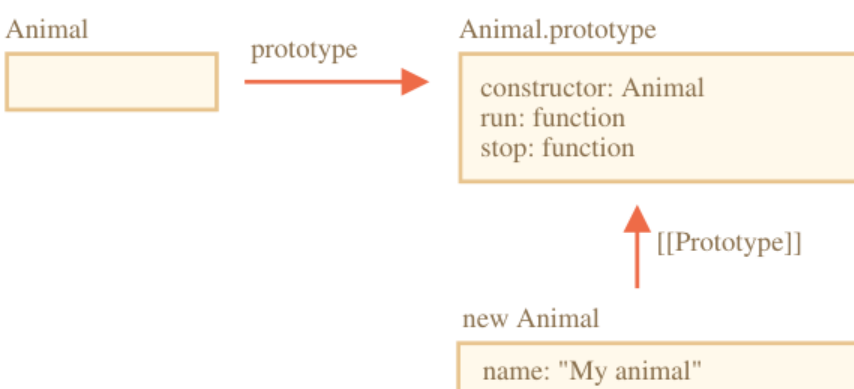
К решению

Наследование классов

Допустим, у нас есть два класса.

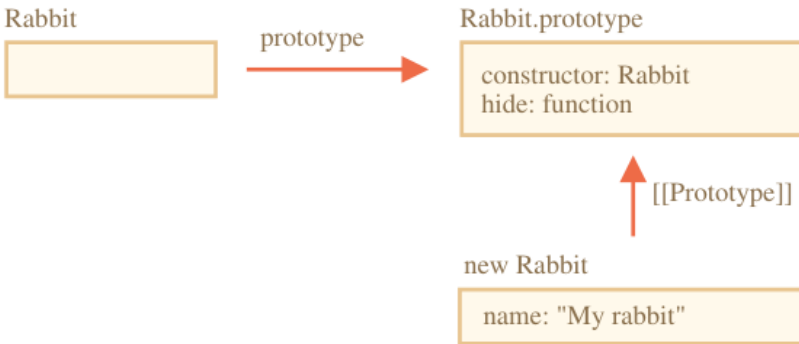
`Animal`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} стоит.`);
  }
}

let animal = new Animal("Мой питомец");
```



...И `Rabbit`:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbit = new Rabbit("Мой кролик");
```



Сейчас они полностью независимы.

Но мы хотим, чтобы `Rabbit` расширял `Animal`. Другими словами, кролики должны происходить от животных, т.е. иметь доступ к методам `Animal` и расширять функциональность `Animal` своими методами.

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово `"extends"` и указать название родительского класса перед `{..}`.

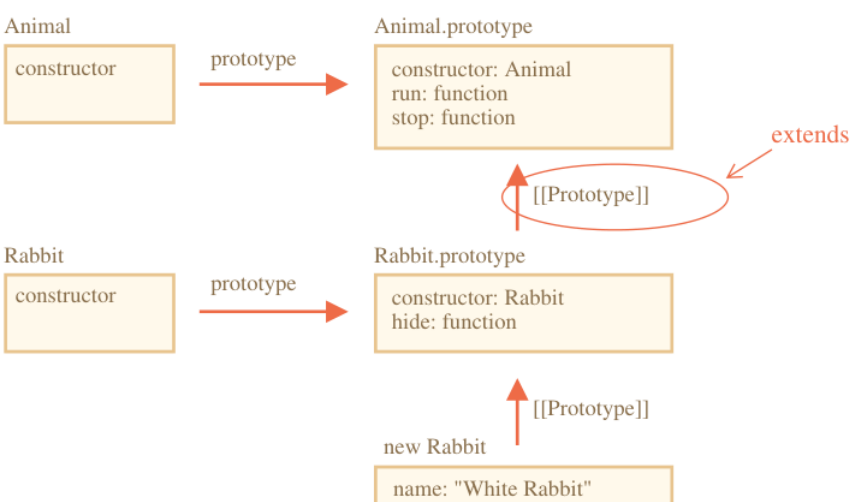
Ниже `Rabbit` наследует от `Animal`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }
```

```
  stop() {
    this.speed = 0;
    alert(`${this.name} стоит.`);
  }
}

// Наследуем от Animal указывая "extends Animal"

  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.hide(); // Белый кролик прячется!
class Rabbit extends Animal {
```

Теперь код `Rabbit` стал короче, так как используется конструктор класса `Animal` по умолчанию и кролик может использовать метод `run` как и все животные.

Ключевое слово `extends` работает, используя прототипы. Оно устанавливает `Rabbit.prototype.[[Prototype]]` в `Animal.prototype`. Так что если метод не найден в `Rabbit.prototype`, JavaScript берёт его из `Animal.prototype`.



Как мы помним из главы Встроенные прототипы, в JavaScript используется наследование на прототипах для встроенных объектов. Например `Date.prototype.[[Prototype]]` это `Object.prototype`, поэтому у дат есть универсальные методы объекта.

После `extends` разрешены любые выражения

Синтаксис создания класса допускает указывать после `extends` не только класс, но любое выражение.

Пример вызова функции, которая генерирует родительский класс:

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}


new User().sayHi(); // Привет
class User extends f("Привет") {}
```

Здесь `class User` наследует от результата вызова `f("Привет")`.

Это может быть полезно для продвинутых приёмов проектирования, где мы можем использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

Переопределение методов

Давайте пойдём дальше и переопределим метод. Сейчас `Rabbit` наследует от `Animal` метод `stop`, который устанавливает `this.speed = 0`.

Если мы определим свой метод `stop` в классе `Rabbit`, то он будет использоваться взамен родительского:

```
class Rabbit extends Animal {
  stop() {
    // ...будет использован для rabbit.stop()
  }
}
```

…Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово `"super"` для таких случаев.

- `super.method(...)` вызывает родительский метод.
- `super(...)` вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  run(speed) {
    this.speed = speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }

  stop() {
    this.speed = 0;
    alert(`${this.name} стоит.`);
```

```
  }

}

class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }


}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.
rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!
  stop() {
    super.stop(); // вызываем родительский
метод stop
    this.hide(); // и затем hide
  }
```

Теперь у класса `Rabbit` есть метод `stop`, который вызывает родительский `super.stop()` в процессе выполнения.

У стрелочных функций нет `super`

Как упоминалось в главе Повторяем стрелочные функции, стрелочные функции не имеют `super`.

При обращении к `super` стрелочной функции он берётся из внешней функции:

```
class Rabbit extends Animal {
  stop() {
    setTimeout(() => super.stop(), 1000); //
вызывает родительский stop после 1 секунды
  }
}
```

В примере `super` в стрелочной функции тот же

самый, что и в `stop()`, поэтому метод отрабатывает как и ожидается. Если бы мы указали здесь «обычную» функцию, была бы ошибка:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

Переопределение конструктора

С конструкторами немного сложнее.

До сих пор у `Rabbit` не было своего конструктора.

Согласно спецификации, если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор:

```
class Rabbit extends Animal {
  // генерируется для классов-потомков, у которых нет своего конструктора

}
```

```
  constructor(...args) {
    super(...args);
  }
```

Как мы видим, он просто вызывает конструктор родительского класса. Так будет происходить, пока мы не создадим собственный конструктор.

Давайте добавим конструктор для `Rabbit`. Он будет устанавливать `earLength` в дополнение к `name`:

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
```

```
  // ...
}

class Rabbit extends Animal {


  // ...
}

  constructor(name, earLength) {
    this.speed = 0;
    this.name = name;
    this.earLength = earLength;
  }
// Не работает!
let rabbit = new Rabbit("Белый кролик", 10);
// Error: this is not defined.
```

Упс! При создании кролика – ошибка! Что не так?

Если коротко, то в классах-потомках конструктор обязан вызывать `super(...)`, и (!) делать это перед использованием `this`.

…Но почему? Что происходит? Это требование кажется довольно странным.

Конечно, всему есть объяснение. Давайте углубимся в детали, чтобы вы действительно поняли, что происходит.

В JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством `[[ConstructorKind]]:"derived"`.

Разница в следующем:

- Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его `this`.
- Когда запускается конструктор унаследованного

класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

Поэтому, если мы создаём собственный конструктор, мы должны вызвать `super`, в противном случае объект для `this` не будет создан, и мы получим ошибку.

Чтобы конструктор `Rabbit` работал, он должен вызвать `super()` до того, как использовать `this`, чтобы не было ошибки:

```
class Animal {

  constructor(name) {
    this.speed = 0;
    this.name = name;
  }

  // ...
}

class Rabbit extends Animal {

  constructor(name, earLength) {

    this.earLength = earLength;
  }

  // ...
}
```

```
    super(name);
// теперь работает
let rabbit = new Rabbit("Белый кролик", 10);
alert(rabbit.name); // Белый кролик
alert(rabbit.earLength); // 10
```

Устройство super, [[HomeObject]]

Продвинутая информация

Если вы читаете учебник первый раз – эту секцию можно пропустить.

Она рассказывает о внутреннем устройстве наследования и вызов `super`.

Давайте заглянем «под капот» `super`. Здесь есть некоторые интересные моменты.

Вообще, исходя из наших знаний до этого момента, `super` вообще не может работать!

Ну правда, давайте спросим себя – как он должен работать, чисто технически? Когда метод объекта выполняется, он получает текущий объект как `this`. Если мы вызываем `super.method()`, то движку необходимо получить `method` из прототипа текущего объекта. И как ему это сделать?

Задача может показаться простой, но это не так. Движок знает текущий `this` и мог бы попытаться получить родительский метод как `this.__proto__.method`. Однако, увы, такой «наивный» путь не работает.

Продемонстрируем проблему. Без классов, используя простые объекты для наглядности.

Вы можете пропустить эту часть и перейти ниже к подсекции `[[HomeObject]]`, если не хотите знать детали. Вреда не будет. Или читайте далее, если хотите разобраться.

В примере ниже `rabbit.__proto__ = animal`. Попробуем в `rabbit.eat()` вызвать `animal.eat()`, используя `this.__proto__`:

```
let animal = {
  name: "Animal",
  eat() {
    alert(`${this.name} ест.`);
  }
```

```
};

let rabbit = {
  __proto__: animal,
  name: "Кролик",
  eat() {

  }
};

rabbit.eat(); // Кролик ест.
    // вот как предположительно может работать
super.eat()
    this.__proto__.eat.call(this); // (*)
```

В строке `(*)` мы берём `eat` из прототипа (`animal`) и вызываем его в контексте текущего объекта. Обратите внимание, что `.call(this)` здесь неспроста: простой вызов `this.__proto__.eat()` будет выполнять родительский `eat` в контексте прототипа, а не текущего объекта.

Приведённый выше код работает так, как задумано: выполняется нужный `alert`.

Теперь давайте добавим ещё один объект в цепочку наследования и увидим, как все сломается:

```
let animal = {
  name: "Животное",
  eat() {
    alert(`${this.name} ест.`);
  }
};

let rabbit = {
  __proto__: animal,
  eat() {
    // ...делаем что-то специфичное для кролика и вызываем
родительский (animal) метод
    this.__proto__.eat.call(this); // (*)
  }
};

let longEar = {
  __proto__: rabbit,
```

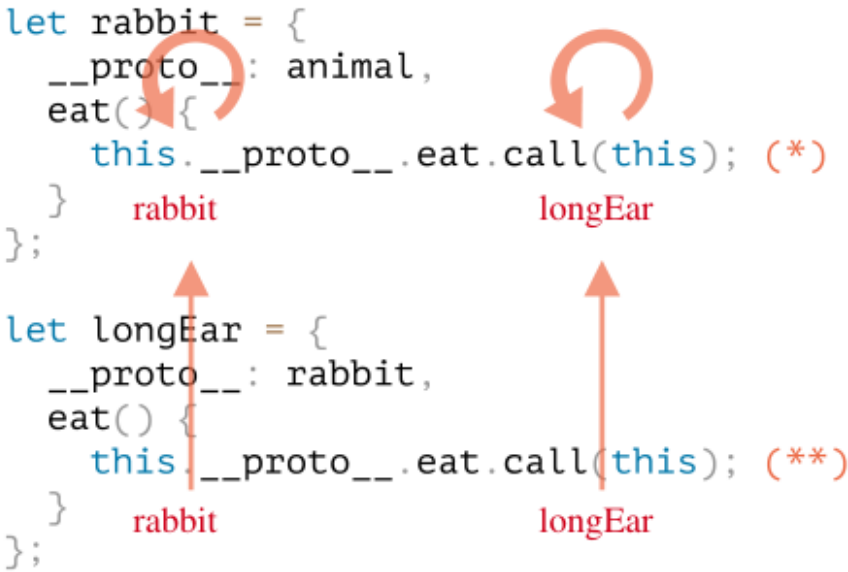
```
  eat() {
    // ...делаем что-то, связанное с длинными ушами и вызываем
родительский (rabbit) метод
    this.__proto__.eat.call(this); // (**)
  }
};

longEar.eat(); // Error: Maximum call stack
size exceeded
```

Теперь код не работает! Ошибка возникает при попытке вызова `longEar.eat()`.

На первый взгляд все не так очевидно, но если мы проследим вызов `longEar.eat()`, то сможем понять причину ошибки. В обеих строках `(*)` и `(**)` значение `this` – это текущий объект (`longEar`). Это важно: для всех методов объекта `this` указывает на текущий объект, а не на прототип или что-то ещё.

Итак, в обеих линиях `(*)` и `(**)` значение `this.__proto__` одно и то же: `rabbit`. В обоих случаях метод `rabbit.eat` вызывается в бесконечном цикле не поднимаясь по цепочке вызовов.

Картина того, что происходит:



1. Внутри `longEar.eat()` строка `(**)` вызывает `rabbit.eat` со значением `this=longEar`.

```
// внутри longEar.eat() у нас this = longEar
this.__proto__.eat.call(this) // (**)
// становится
longEar.__proto__.eat.call(this)
// то же что и
rabbit.eat.call(this);
```

2. В строке `(*)` в `rabbit.eat` мы хотим передать вызов выше по цепочке, но `this=longEar`, поэтому `this.__proto__.eat` снова равен `rabbit.eat`!

```
// внутри rabbit.eat() у нас также this = longEar
this.__proto__.eat.call(this) // (*)
// становится
longEar.__proto__.eat.call(this)
// или (снова)
rabbit.eat.call(this);
```

3. ...`rabbit.eat` вызывает себя в бесконечном цикле, потому что не может подняться дальше по цепочке.

Проблема не может быть решена с помощью одного только `this`.

`[[HomeObject]]`

Для решения этой проблемы в JavaScript было добавлено специальное внутреннее свойство для функций: `[[HomeObject]]`.

Когда функция объявлена как метод внутри класса или объекта, её свойство `[[HomeObject]]` становится равно этому объекту.

Затем `super` использует его, чтобы получить прототип родителя и его методы.

Давайте посмотрим, как это работает – опять же, используя простые объекты:

```
let animal = {
  name: "Животное",
  eat() {         // animal.eat.[[HomeObject]] == animal
    alert(`${this.name} ест.`);
  }
};

let rabbit = {
  __proto__: animal,
  name: "Кролик",
  eat() {         // rabbit.eat.[[HomeObject]] == rabbit
    super.eat();
  }
};

let longEar = {
  __proto__: rabbit,
  name: "Длинноух",
  eat() {         // longEar.eat.[[HomeObject]] == longEar
    super.eat();
  }
};

// работает верно
longEar.eat();  // Длинноух ест.
```

Это работает как задумано благодаря `[[HomeObject]]`.

Метод, такой как `longEar.eat`, знает свой `[[HomeObject]]` и получает метод родителя из его прототипа. Вообще без использования `this`.

Методы не «свободны»

До этого мы неоднократно видели, что функции в JavaScript «свободны», не привязаны к объектам. Их можно копировать между объектами и вызывать с любым `this`.

Но само существование `[[HomeObject]]` нарушает этот принцип, так как методы запоминают свои объекты. `[[HomeObject]]` нельзя изменить, эта связь – навсегда.

Единственное место в языке, где используется `[[HomeObject]]` – это `super`. Поэтому если метод не использует `super`, то мы все ещё можем считать его свободным и копировать между объектами. А вот если `super` в коде есть, то возможны побочные эффекты.

Вот пример неверного результата `super` после копирования:

```
let animal = {
  sayHi() {
    console.log("Я животное");
  }
};

// rabbit наследует от animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

let plant = {
  sayHi() {
    console.log("Я растение");
  }
};

// tree наследует от plant
let tree = {
```

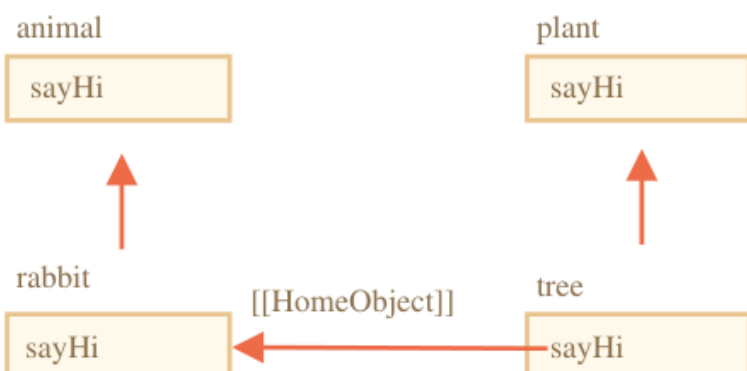
```
  __proto__: plant,

};

  sayHi: rabbit.sayHi // (*)
tree.sayHi();  // Я животное (?!?)
```

Вызов `tree.sayHi()` показывает «Я животное». Определённо неверно.

Причина проста:

> В строке `(*)`, метод `tree.sayHi` скопирован из `rabbit`. Возможно, мы хотели избежать дублирования кода?
>
> Его `[[HomeObject]]` – это `rabbit`, ведь он был создан в `rabbit`. Свойство `[[HomeObject]]` никогда не меняется.
>
> В коде `tree.sayHi()` есть вызов `super.sayHi()`. Он идёт вверх от `rabbit` и берёт метод из `animal`.

Вот диаграмма происходящего:



Методы, а не свойства-функции

Свойство `[[HomeObject]]` определено для методов как классов, так и обычных объектов. Но для объектов методы должны быть объявлены именно как `method()`, а не

"method: function()".

Для нас различий нет, но они есть для JavaScript.

В приведённом ниже примере используется синтаксис не метода, свойства-функции. Поэтому у него нет `[[HomeObject]]`, и наследование не работает:

```
let animal = {
  eat: function() { // намеренно пишем так, а не eat() { ...
    // ...
  }
};

let rabbit = {
  __proto__: animal,
  eat: function() {
    super.eat();
  }
};

rabbit.eat();  // Ошибка вызова super (потому что нет [[HomeObject]])
```

Итого

1. Чтобы унаследовать от класса: `class Child extends Parent`:

   При этом `Child.prototype.__proto__` будет равен `Parent.prototype`, так что методы будут унаследованы.

3. При переопределении конструктора:

   Обязателен вызов конструктора родителя `super()` в конструкторе `Child` до обращения к `this`.

5. При переопределении другого метода:

   Мы можем вызвать `super.method()` в методе `Child` для обращения к методу родителя `Parent`.

7. Внутренние детали:

   Методы запоминают свой объект во внутреннем

свойстве `[[HomeObject]]`. Благодаря этому работает `super`, он в его прототипе ищет родительские методы.

Поэтому копировать метод, использующий `super`, между разными объектами небезопасно.

Также:

У функций-стрелок нет своего `this` и `super`, поэтому они «прозрачно» встраиваются во внешний контекст.

Задачи

Ошибка создания экземпляра класса

важность: 5

В коде ниже класс `Rabbit` наследует `Animal`.

К сожалению, объект класса `Rabbit` не создаётся. Что не так? Исправьте ошибку.

```
class Animal {

  constructor(name) {
    this.name = name;
  }

}

class Rabbit extends Animal {
  constructor(name) {
    this.name = name;
    this.created = Date.now();
  }
}


alert(rabbit.name);
```

```
let rabbit = new Rabbit("Белый кролик"); //
Error: this is not defined
```

К решению

Улучшенные часы

важность: 5

У нас есть класс `Clock`. Сейчас он выводит время каждую секунду

```
class Clock {
  constructor({ template }) {
    this.template = template;
  }

  render() {
    let date = new Date();

    let hours = date.getHours();
    if (hours < 10) hours = '0' + hours;

    let mins = date.getMinutes();
    if (mins < 10) mins = '0' + mins;

    let secs = date.getSeconds();
    if (secs < 10) secs = '0' + secs;

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs);

    console.log(output);
  }

  stop() {
    clearInterval(this.timer);
  }

  start() {
    this.render();
    this.timer = setInterval(() => this.render(), 1000);
  }
}
```

Создайте новый класс `ExtendedClock`, который будет наследоваться от `Clock` и добавьте параметр `precision` – количество миллисекунд между «тиками». Установите значение в `1000` (1 секунда) по умолчанию.

> Сохраните ваш код в файл `extended-clock.js`
>
> Не изменяйте класс `clock.js`. Расширьте его.

Открыть песочницу для задачи.

К решению

Статические свойства и методы

Мы также можем присвоить метод самой функции-классу, а не её `"prototype"`. Такие методы называются *статическими*.

В классе такие методы обозначаются ключевым словом `static`, например:

```
class User {

    alert(this === User);
  }
}

User.staticMethod(); // true
  static staticMethod() {
```

Это фактически то же самое, что присвоить метод напрямую как свойство функции:

```
class User { }

User.staticMethod = function() {
  alert(this === User);
};
```

Значением `this` при вызове `User.staticMethod()` является сам конструктор класса `User` (правило «объект до точки»).

Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам.

Например, есть объекты статей `Article`, и нужна функция для их сравнения. Естественное решение – сделать для этого метод `Article.compare`:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }


}

// использование
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];



alert( articles[0].title ); // CSS
  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
articles.sort(Article.compare);
```

Здесь метод `Article.compare` стоит «над» статьями, как способ их сравнения. Это метод не отдельной статьи, а

всего класса.

Другим примером может быть так называемый «фабричный» метод. Представим, что нам нужно создавать статьи различными способами:

1. Создание через заданные параметры (`title`, `date` и т. д.).
2. Создание пустой статьи с сегодняшней датой.
3. …или как-то ещё.

Первый способ может быть реализован через конструктор. А для второго можно использовать статический метод класса.

Такой как `Article.createTodays()` в следующем примере:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }


}

let article = Article.createTodays();

alert( article.title ); // Сегодняшний дайджест
  static createTodays() {
    // помним, что this = Article
    return new this("Сегодняшний дайджест",
new Date());
  }
```

Теперь каждый раз, когда нам нужно создать сегодняшний дайджест, нужно вызывать `Article.createTodays()`. Ещё раз, это не метод одной статьи, а метод всего класса.

Статические методы также используются в классах,

относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных, например:

```
// предположим, что Article - это специальный класс для управления статьями
// статический метод для удаления статьи:
Article.remove({id: 12345});
```

Статические свойства

> **Новая возможность**
>
> Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome.

Статические свойства также возможны, они выглядят как свойства класса, но с `static` в начале:

```
class Article {
  static publisher = "Илья Кантор";
}

alert( Article.publisher ); // Илья Кантор
```

Это то же самое, что и прямое присваивание `Article`:

```
Article.publisher = "Илья Кантор";
```

Наследование статических свойств и методов

Статические свойства и методы наследуются.

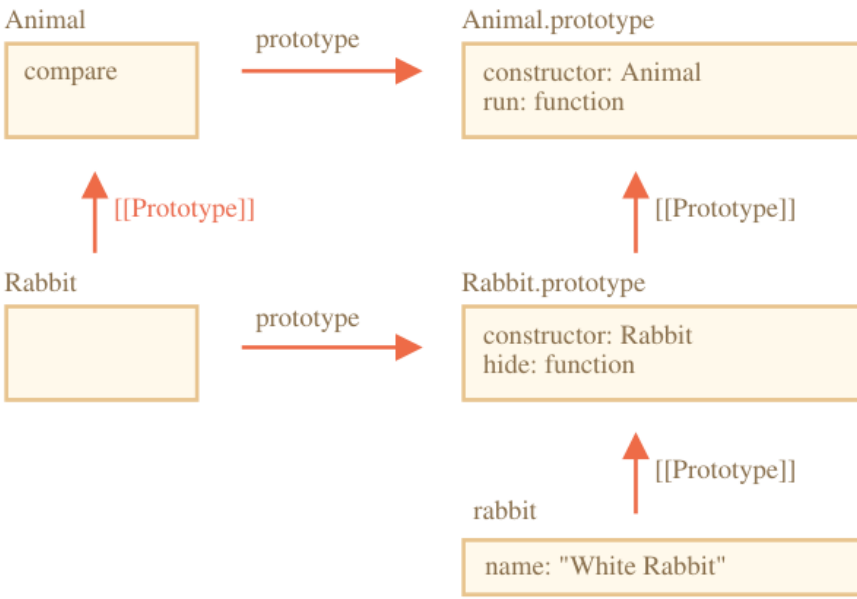
Например, метод `Animal.compare` в коде ниже наследуется и доступен как `Rabbit.compare`:

```
class Animal {

  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }

  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} бежит со скоростью ${this.speed}.`);
  }



}

// Наследует от Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} прячется!`);
  }
}

let rabbits = [
  new Rabbit("Белый кролик", 10),
  new Rabbit("Чёрный кролик", 5)
];



rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.
  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
rabbits.sort(Rabbit.compare);
```

Мы можем вызвать `Rabbit.compare`, при этом будет

вызван унаследованный `Animal.compare`.

Как это работает? Снова с использованием прототипов. Как вы уже могли предположить, `extends` даёт `Rabbit` ссылку `[[Prototype]]` на `Animal`.



Так что `Rabbit extends Animal` создаёт две ссылки на прототип:

1. Функция `Rabbit` прототипно наследует от функции `Animal`.
2. `Rabbit.prototype` прототипно наследует от `Animal.prototype`.

В результате наследование работает как для обычных, так и для статических методов.

Давайте это проверим кодом:

```
class Animal {}
class Rabbit extends Animal {}

// для статики
```

```
alert(Rabbit.__proto__ === Animal); // true

// для обычных методов
alert(Rabbit.prototype.__proto__ === Animal.prototype); //
true
```

Итого

Статические методы используются для функциональности, принадлежат классу «в целом», а не относятся к конкретному объекту класса.

Например, метод для сравнения двух статей `Article.compare(article1, article2)` или фабричный метод `Article.createTodays()`.

В объявлении класса они помечаются ключевым словом `static`.

Статические свойства используются в тех случаях, когда мы хотели бы сохранить данные на уровне класса, а не какого-то одного объекта.

Синтаксис:

```
class MyClass {
  static property = ...;

  static method() {
    ...
  }
}
```

Технически, статическое объявление – это то же самое, что и присвоение классу:

```
MyClass.property = ...
MyClass.method = ...
```

Статические свойства и методы наследуются.

Для `class B extends A` прототип класса `B` указывает на `A`: `B.[[Prototype]] = A`. Таким образом, если поле не найдено в `B`, поиск продолжается в `A`.

Задачи

Класс расширяет объект?

важность: 5

Как мы уже знаем, все объекты наследуют от `Object.prototype` и имеют доступ к «общим» методам объекта, например `hasOwnProperty`.

Пример:

```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Rab");

// метод hasOwnProperty от Object.prototype
alert( rabbit.hasOwnProperty('name') ); // true
```

Но что если мы явно напишем `"class Rabbit extends Object"` – тогда результат будет отличаться от обычного `"class Rabbit"`?

В чем разница?

Ниже пример кода с таким наследованием (почему он не работает? исправьте его):

```
class Rabbit extends Object {
  constructor(name) {
    this.name = name;
  }
}

let rabbit = new Rabbit("Кроль");

alert( rabbit.hasOwnProperty('name') ); // Ошибка
```

К решению

Приватные и защищённые методы и свойства

Один из важнейших принципов объектно-ориентированного программирования – разделение внутреннего и внешнего интерфейсов.

Это обязательная практика в разработке чего-либо сложнее, чем «hello world».

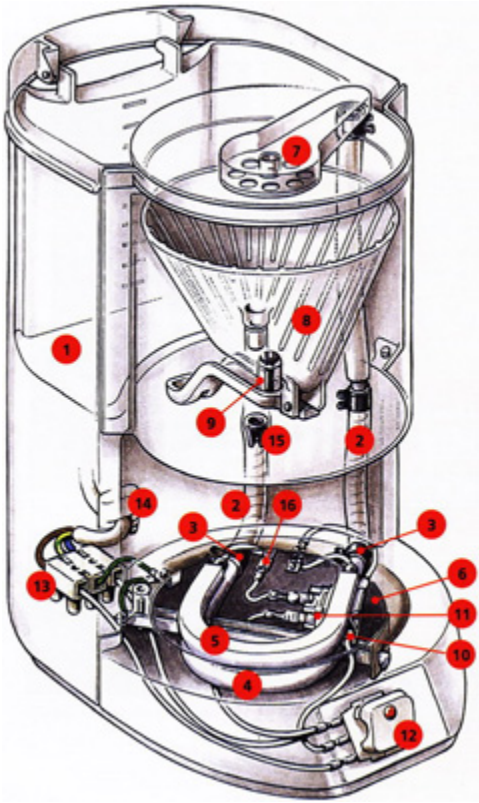
Чтобы понять этот принцип, давайте на секунду забудем о программировании и обратим взгляд на реальный мир.

Устройства, которыми мы пользуемся, обычно довольно сложно устроены. Но разделение внутреннего и внешнего интерфейсов позволяет нам пользоваться ими без каких-либо проблем.

Пример из реальной жизни

Например, кофеварка. Простая снаружи: кнопка, экран, несколько отверстий… И, конечно, результат – прекрасный кофе! :)

Но внутри… (картинка из инструкции по ремонту)



Множество деталей. Но мы можем пользоваться ею, ничего об этом не зная.

Кофеварки довольно надёжны, не так ли? Мы можем пользоваться ими годами, и если что-то пойдёт не так – отнесём в ремонт.

Секрет надёжности и простоты кофеварки – все детали хорошо отлажены и *спрятаны* внутри.

Если мы снимем защитный кожух с кофеварки, то пользоваться ею будет гораздо сложнее (куда нажимать?) и опаснее (может привести к поражению электрическим током).

Как мы увидим, в программировании объекты похожи на кофеварки.

Но, чтобы скрыть внутренние детали, мы будем

использовать не защитный кожух, а специальный синтаксис языка и соглашения.

Внутренний и внешний интерфейсы

В объектно-ориентированном программировании свойства и методы разделены на 2 группы:

Внутренний интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса.

Внешний интерфейс – методы и свойства, доступные снаружи класса.

Если мы продолжаем аналогию с кофеваркой – то, что скрыто внутри: трубка кипятильника, нагревательный элемент и т.д. – это внутренний интерфейс.

Внутренний интерфейс используется для работы объекта, его детали используют друг друга. Например, трубка кипятильника прикреплена к нагревательному элементу.

Но снаружи кофеварка закрыта защитным кожухом, так что никто не может добраться до сложных частей. Детали скрыты и недоступны. Мы можем использовать их функции через внешний интерфейс.

Итак, всё, что нам нужно для использования объекта, это знать его внешний интерфейс. Мы можем совершенно не знать, как это работает внутри, и это здорово.

Это было общее введение.

В JavaScript есть два типа полей (свойств и методов) объекта:

Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы

использовали только публичные свойства и методы.

Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрешён доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что мы обычно хотим, чтобы наследующие классы получали доступ к внутренним полям.

Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

А теперь давайте сделаем кофеварку на JavaScript со всеми этими типами свойств. Кофеварка имеет множество деталей, мы не будем их моделировать для простоты примера (хотя могли бы).

Защищённое свойство «waterAmount»

Давайте для начала создадим простой класс для описания кофеварки:

```
class CoffeeMachine {
  waterAmount = 0; // количество воды внутри

  constructor(power) {
    this.power = power;
    alert( `Создана кофеварка, мощность: ${power}` );
  }

}

// создаём кофеварку
let coffeeMachine = new CoffeeMachine(100);
```

```
// добавляем воды
coffeeMachine.waterAmount = 200;
```

Прямо сейчас свойства `waterAmount` и `power` публичные. Мы можем легко получать и устанавливать им любое значение извне.

Давайте изменим свойство `waterAmount` на защищённое, чтобы иметь больше контроля над ним. Например, мы не хотим, чтобы кто-либо устанавливал его ниже нуля.

Защищённые свойства обычно начинаются с префикса `_`.

Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению.

Так что наше свойство будет называться `_waterAmount`:

```
class CoffeeMachine {
  _waterAmount = 0;

  set waterAmount(value) {
    if (value < 0) throw new Error("Отрицательное количество
воды");
    this._waterAmount = value;
  }

  get waterAmount() {
    return this._waterAmount;
  }

  constructor(power) {
    this._power = power;
  }

}

// создаём новую кофеварку
let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды
coffeeMachine.waterAmount = -10; // Error: Отрицательное
количество воды
```

Теперь доступ под контролем, поэтому указать воду ниже нуля не удалось.

Свойство только для чтения «power»

Давайте сделаем свойство `power` доступным только для чтения. Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось.

Это как раз требуется для кофеварки: мощность никогда не меняется.

Для этого нам нужно создать только геттер, но не сеттер:

```
class CoffeeMachine {
  // ...

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }

}

// создаём кофеварку
let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)
```

Геттеры/сеттеры

Здесь мы использовали синтаксис геттеров/сеттеров.

Но в большинстве случаев использование функций `get.../set...` предпочтительнее:

```
class CoffeeMachine {
  _waterAmount = 0;

    {
     if (value < 0) throw new Error("Отрицательное
количество воды");
     this._waterAmount = value;
   }

    {
     return this._waterAmount;
   }
}

new CoffeeMachine().setWaterAmount(100);
setWaterAmount
(value)
getWaterAmount
()
```

Это выглядит немного длиннее, но функции более гибкие. Они могут принимать несколько аргументов (даже если они нам сейчас не нужны). Итак, на будущее, если нам надо что-то отрефакторить, функции более безопасный выбор.

С другой стороны, синтаксис get/set короче, решать вам.

Защищённые поля наследуются

Если мы унаследуем `class MegaMachine extends CoffeeMachine`, ничто не помешает нам обращаться к `this._waterAmount` или `this._power` из методов нового класса.

Таким образом защищённые методы, конечно же, наследуются. В отличие от приватных полей, в чём мы убедимся ниже.

Приватное свойство «#waterLimit»

Новая возможность

Эта возможность была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифил.

Есть новшество в языке JavaScript, которое почти добавлено в стандарт: оно добавляет поддержку приватных свойств и методов.

Приватные свойства и методы должны начинаться с `#`. Они доступны только внутри класса.

Например, в классе ниже есть приватное свойство `#waterLimit` и приватный метод `#checkWater` для проверки количества воды:

```
class CoffeeMachine {


}

let coffeeMachine = new CoffeeMachine();
```

```
  #waterLimit = 200;
  #checkWater(value) {
    if (value < 0) throw new
Error("Отрицательный уровень воды");
    if (value > this.#waterLimit) throw new
```

```
Error("Слишком много воды");
    }
// снаружи  нет доступа к приватным методам
класса
coffeeMachine.#checkWater(); // Error
coffeeMachine.#waterLimit = 1000; // Error
```

На уровне языка `#` является специальным символом, который означает, что поле приватное. Мы не можем получить к нему доступ извне или из наследуемых классов.

Приватные поля не конфликтуют с публичными. У нас может быть два поля одновременно – приватное `#waterAmount` и публичное `waterAmount`.

Например, давайте сделаем аксессор `waterAmount` для `#waterAmount`:

```
class CoffeeMachine {

  #waterAmount = 0;

  get waterAmount() {
    return this.#waterAmount;
  }

  set waterAmount(value) {
    if (value < 0) throw new Error("Отрицательный уровень
воды");
    this.#waterAmount = value;
  }
}

let machine = new CoffeeMachine();

machine.waterAmount = 100;
alert(machine.#waterAmount); // Error
```

В отличие от защищённых, функциональность приватных полей обеспечивается самим языком. Это хорошо.

Но если мы унаследуем от `CoffeeMachine`, то мы не получим прямого доступа к `#waterAmount`. Мы будем

вынуждены полагаться на геттер/сеттер `waterAmount`:

```
class MegaCoffeeMachine extends CoffeeMachine {
  method() {

  }
}
    alert( this.#waterAmount ); // Error: can only access from CoffeeMachine
```

Во многих случаях такое ограничение слишком жёсткое. Раз уж мы расширяем `CoffeeMachine`, у нас может быть вполне законная причина для доступа к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Важно:

Приватные поля особенные.

Как мы помним, обычно мы можем получить доступ к полям объекта с помощью this[name]:

```
class User {
  ...
  sayHi() {
    let fieldName = "name";
    alert(`Hello, ${}`);
  }
}
this
[fieldName]
```

С приватными свойствами такое невозможно: `this['#name']` не работает. Это ограничение синтаксиса сделано для обеспечения приватности.

Итого

В терминах ООП отделение внутреннего интерфейса от внешнего называется инкапсуляция.

Это даёт следующие выгоды:

Защита для пользователей, чтобы они не выстрелили себе в ногу

Представьте себе, что есть команда разработчиков, использующая кофеварку. Она была изготовлена компанией «Лучшие Кофеварки» и работает нормально, но защитный кожух был снят. Внутренний интерфейс стал доступен извне.

Все разработчики культурны – они используют кофеварку по назначению. Но один из них, Джон, решил, что он самый умный, и сделал некоторые изменения во внутренностях кофеварки. После чего кофеварка вышла из строя через два дня.

Это, конечно, не вина Джона, а скорее человека, который снял защитный кожух и позволил Джону делать свои манипуляции.

То же самое в программировании. Если пользователь класса изменит вещи, не предназначенные для изменения извне – последствия непредсказуемы.

Поддерживаемость

Ситуация в программировании сложнее, чем с реальной кофеваркой, потому что мы не просто покупаем её один раз. Код постоянно подвергается разработке и улучшению.

Если мы чётко отделим внутренний интерфейс, то разработчик класса сможет свободно менять его внутренние свойства и методы, даже не информируя пользователей…

Если вы разработчик такого класса, то приятно знать, что приватные методы можно безопасно переименовывать, их параметры можно изменять и даже удалять, потому что от них не зависит никакой внешний код.

В новой версии вы можете полностью всё переписать, но пользователю будет легко обновиться, если внешний интерфейс остался такой же.

Сокрытие сложности

Люди обожают использовать простые вещи. По крайней мере, снаружи. Что внутри – это другое дело.

Программисты не являются исключением.

Всегда удобно, когда детали реализации скрыты, и доступен простой, хорошо документированный внешний интерфейс.

Для сокрытия внутреннего интерфейса мы используем защищённые или приватные свойства:

> Защищённые поля имеют префикс `_`. Это хорошо известное соглашение, не поддерживаемое на уровне языка. Программисты должны обращаться к полю, начинающемуся с `_`, только из его класса и классов, унаследованных от него.
>
> Приватные поля имеют префикс `#`. JavaScript гарантирует, что мы можем получить доступ к таким полям только внутри класса.

В настоящее время приватные поля не очень хорошо поддерживаются в браузерах, но можно использовать полифил.

Расширение встроенных классов

От встроенных классов, таких как `Array`, `Map` и других, тоже можно наследовать.

Например, в этом примере `PowerArray` наследуется от встроенного `Array`:

```
// добавим один метод (можно более одного)
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

Обратите внимание на интересный момент: встроенные методы, такие как `filter`, `map` и другие возвращают новые объекты унаследованного класса `PowerArray`. Их внутренняя реализация такова, что для этого они используют свойство объекта `constructor`.

В примере выше,

```
arr.constructor === PowerArray
```

Поэтому при вызове метода `arr.filter()` он внутри создаёт массив результатов, именно используя `arr.constructor`, а не обычный массив. Это замечательно, поскольку можно продолжать использовать методы `PowerArray` далее на результатах.

Более того, мы можем настроить это поведение.

При помощи специального статического геттера `Symbol.species` можно вернуть конструктор, который JavaScript будет использовать в `filter`, `map` и других методах для создания новых объектов.

Если бы мы хотели, чтобы методы `map`, `filter` и т. д. возвращали обычные массивы, мы могли бы вернуть `Array` в `Symbol.species`, вот так:

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }


}

let arr = new PowerArray(1, 2, 5, 10, 50);
alert(arr.isEmpty()); // false

// filter создаст новый массив, используя
arr.constructor[Symbol.species] как конструктор
let filteredArr = arr.filter(item => item >= 10);


alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is
not a function
  // встроенные методы массива будут
использовать этот метод как конструктор
  static get [Symbol.species]() {
    return Array;
  }
// filteredArr не является PowerArray, это
Array
```

Как вы видите, теперь `.filter` возвращает `Array`. Расширенная функциональность не будет передаваться далее.

Аналогично работают другие коллекции

Другие коллекции, такие как `Map`, `Set`, работают аналогично. Они также используют `Symbol.species`.

Отсутствие статического наследования встроенных классов

У встроенных объектов есть собственные статические методы, например `Object.keys`, `Array.isArray` и т. д.

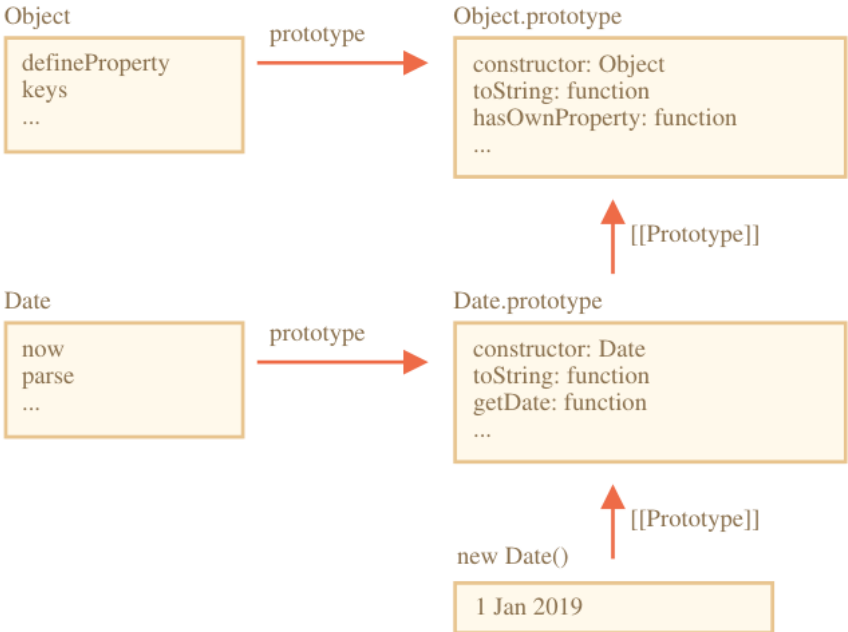
Как мы уже знаем, встроенные классы расширяют друг друга.

Обычно, когда один класс наследует другой, то наследуются и статические методы. Это было подробно разъяснено в главе Статические свойства и методы.

Но встроенные классы – исключение. Они не наследуют статические методы друг друга.

Например, и `Array`, и `Date` наследуют от `Object`, так что в их экземплярах доступны методы из `Object.prototype`. Но `Array.[[Prototype]]` не ссылается на `Object`, поэтому нет методов `Array.keys()` или `Date.keys()`.

Ниже вы видите структуру `Date` и `Object`:



Как видите, нет связи между `Date` и `Object`. Они независимы, только `Date.prototype` наследует от `Object.prototype`.

В этом важное отличие наследования встроенных объектов от того, что мы получаем с использованием `extends`.

Проверка класса: "instanceof"

Оператор `instanceof` позволяет проверить, к какому классу принадлежит объект, с учётом наследования.

Такая проверка может потребоваться во многих случаях. Здесь мы используем её для создания *полиморфной* функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

Оператор instanceof

Синтаксис:

```
obj instanceof Class
```

Оператор вернёт `true`, если `obj` принадлежит классу `Class` или наследующему от него.

Например:

```
class Rabbit {}
let rabbit = new Rabbit();

// это объект класса Rabbit?
alert( rabbit instanceof Rabbit ); // true
```

Также это работает с функциями-конструкторами:

```
alert( new Rabbit() instanceof Rabbit ); // true
// вместо класса
function Rabbit() {}
```

...И для встроенных классов, таких как `Array`:

```
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

Пожалуйста, обратите внимание, что `arr` также принадлежит классу `Object`, потому что `Array` наследует от `Object`.

Обычно оператор `instanceof` просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода `Symbol.hasInstance`.

Алгоритм работы `obj instanceof Class` работает

примерно так:

1. Если имеется статический метод `Symbol.hasInstance`, тогда вызвать его: `Class[Symbol.hasInstance](obj)`. Он должен вернуть либо `true`, либо `false`, и это конец. Это как раз и есть возможность ручной настройки `instanceof`.

   Пример:

   ```
   // проверка instanceof будет полагать,
   // что всё со свойством canEat - животное Animal
   class Animal {
     static [Symbol.hasInstance](obj) {
       if (obj.canEat) return true;
     }
   }

   let obj = { canEat: true };
   alert(obj instanceof Animal); // true: вызван
   Animal[Symbol.hasInstance](obj)
   ```

2. Большая часть классов не имеет метода `Symbol.hasInstance`. В этом случае используется стандартная логика: проверяется, равен ли `Class.prototype` одному из прототипов в прототипной цепочке `obj`.

   Другими словами, сравнивается:

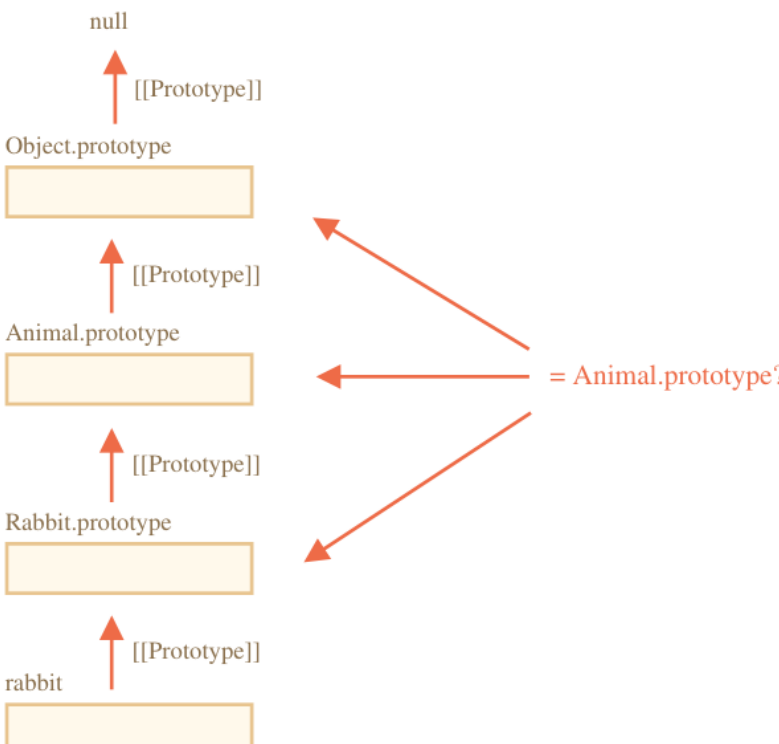
   ```
   obj.__proto__ === Class.prototype?
   obj.__proto__.__proto__ === Class.prototype?
   obj.__proto__.__proto__.__proto__ === Class.prototype?
   ...
   // если какой-то из ответов true - возвратить true
   // если дошли до конца цепочки - false
   ```

   В примере выше `rabbit.__proto__ === Rabbit.prototype`, так что результат будет получен немедленно.

В случае с наследованием, совпадение будет на втором шаге:

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();


// rabbit.__proto__ === Rabbit.prototype
alert(rabbit instanceof Animal); // true
// rabbit.__proto__.__proto__ ===
Animal.prototype (совпадение!)
```

Вот иллюстрация того как `rabbit instanceof Animal` сравнивается с `Animal.prototype`:



Кстати, есть метод objA.isPrototypeOf(objB), который возвращает `true`, если объект `objA` есть где-то в

прототипной цепочке объекта `objB`. Так что `obj instanceof Class` можно перефразировать как `Class.prototype.isPrototypeOf(obj)`.

Забавно, но сам конструктор `Class` не участвует в процессе проверки! Важна только цепочка прототипов `Class.prototype`.

Это может приводить к интересным последствиям при изменении свойства `prototype` после создания объекта.

Как, например, тут:

```
function Rabbit() {}
let rabbit = new Rabbit();

// заменяем прототип
Rabbit.prototype = {};

// ...больше не rabbit!
alert( rabbit instanceof Rabbit ); // false
```

Бонус: Object.prototype.toString возвращает тип

Мы уже знаем, что обычные объекты преобразуется к строке как `[object Object]`:

```
let obj = {};

alert(obj); // [object Object]
alert(obj.toString()); // то же самое
```

Так работает реализация метода `toString`. Но у `toString` имеются скрытые возможности, которые делают метод гораздо более мощным. Мы можем использовать его как расширенную версию `typeof` и как альтернативу `instanceof`.

Звучит странно? Так и есть. Давайте развеем мистику.

Согласно спецификации встроенный метод `toString` может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

- Для числа это будет `[object Number]`
- Для булева типа это будет `[object Boolean]`
- Для `null`: `[object Null]`
- Для `undefined`: `[object Undefined]`
- Для массивов: `[object Array]`
- …и т.д. (поведение настраивается).

Давайте продемонстрируем:

```
// скопируем метод toString в переменную для удобства
let objectToString = Object.prototype.toString;

// какой это тип?
let arr = [];

alert( objectToString.call(arr) ); // [object ]
Array
```

В примере мы использовали call, как описано в главе Декораторы и переадресация вызова, call/apply, чтобы выполнить функцию `objectToString` в контексте `this=arr`.

Внутри, алгоритм метода `toString` анализирует контекст вызова `this` и возвращает соответствующий результат. Больше примеров:

```
let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]
alert( s.call(null) ); // [object Null]
alert( s.call(alert) ); // [object Function]
```

Symbol.toStringTag

Поведение метода объектов `toString` можно настраивать, используя специальное свойство объекта `Symbol.toStringTag`.

Например:

```
let user = {
  [Symbol.toStringTag]: "User"
};

alert( {}.toString.call(user) ); // [object User]
```

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

```
// toStringTag для браузерного объекта и класса
alert( window[Symbol.toStringTag]); // window
alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); //
XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]
alert( {}.toString.call(new XMLHttpRequest()) ); // [object
XMLHttpRequest]
```

Как вы можете видеть, результат – это значение `Symbol.toStringTag` (если он имеется) обёрнутое в `[object ...]`.

В итоге мы получили «typeof на стероидах», который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать `{}.toString.call` вместо `instanceof` для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.

Итого

Давайте обобщим, какие методы для проверки типа мы знаем:

	работает для	возвращает
`typeof`	примитивов	строка
`{}.toString`	примитивов, встроенных объектов, объектов с `Symbol.toStringTag`	строка
`instanceof`	объектов	true/false

Как мы можем видеть, технически `{}.toString` «более продвинут», чем `typeof`.

А оператор `instanceof` – отличный выбор, когда мы работаем с иерархией классов и хотим делать проверки с учётом наследования.

Задачи

Странный instanceof

важность: 5

Почему `instanceof` в примере ниже возвращает `true`? Мы же видим, что `a` не создан с помощью `B()`.

```
function A() {}
function B() {}

A.prototype = B.prototype = {};

let a = new A();

alert( a instanceof B ); // true
```

К решению

Примеси

В JavaScript можно наследовать только от одного объекта. Объект имеет единственный `[[Prototype]]`. И класс может расширить только один другой класс.

Иногда это может ограничивать нас. Например, у нас есть класс `StreetSweeper` и класс `Bicycle`, а мы хотим создать их смесь: `StreetSweepingBicycle`.

Или у нас есть класс `User`, который реализует пользователей, и класс `EventEmitter`, реализующий события. Мы хотели бы добавить функциональность класса `EventEmitter` к `User`, чтобы пользователи могли легко генерировать события.

Для таких случаев существуют «примеси».

По определению из Википедии, примесь – это класс, методы которого предназначены для использования в других классах, причём без наследования от примеси.

Другими словами, *примесь* определяет методы, которые реализуют определённое поведение. Мы не используем примесь саму по себе, а используем её, чтобы добавить функциональность другим классам.

Пример примеси

Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть легко добавлены в прототип любого класса.

В примере ниже примесь `sayHiMixin` имеет методы, которые придают объектам класса `User` возможность вести разговор:

```
let sayHiMixin = {
  sayHi() {
    alert(`Привет, ${this.name}`);
  },
  sayBye() {
    alert(`Пока, ${this.name}`);
  }
};


class User {
  constructor(name) {
    this.name = name;
  }
}

// копируем методы
Object.assign(User.prototype, sayHiMixin);

// теперь User может сказать Привет
new User("Вася").sayHi(); // Привет, Вася!
// примесь
// использование:
```

Это не наследование, а просто копирование методов. Таким образом, класс `User` может наследовать от другого класса, но при этом также включать в себя примеси, «подмешивающие» другие методы, например:

```
class User extends Person {
// ...
}

Object.assign(User.prototype, sayHiMixin);
```

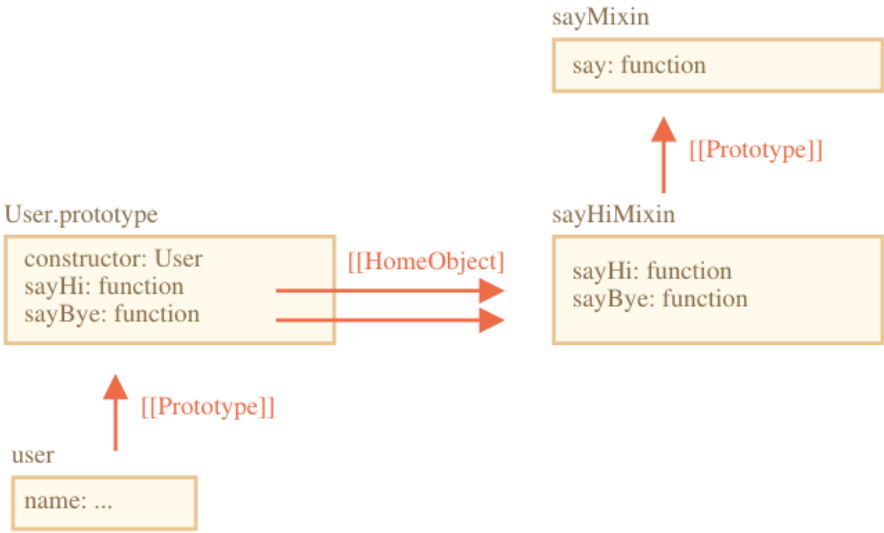
Примеси могут наследовать друг друга.

В примере ниже `sayHiMixin` наследует от `sayMixin`:

```
let sayMixin = {
  say(phrase) {
    alert(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin, // (или мы можем использовать
Object.create для задания прототипа)

  sayHi() {

    super.say(`Привет, ${this.name}`); // (*)
  },
  sayBye() {
    super.say(`Пока, ${this.name}`); // (*)
  }
};

class User {
  constructor(name) {
    this.name = name;
  }
}

// копируем методы
Object.assign(User.prototype, sayHiMixin);

// теперь User может сказать Привет
new User("Вася").sayHi(); // Привет, Вася!
     // вызываем метод родителя
```

Обратим внимание, что при вызове родительского метода `super.say()` из `sayHiMixin` (строки, помеченные `(*)`) этот метод ищется в прототипе самой примеси, а не класса.

Вот диаграмма (см правую часть):



Это связано с тем, что методы `sayHi` и `sayBye` были изначально созданы в объекте `sayHiMixin`. Несмотря на то, что они скопированы, их внутреннее свойство `[[HomeObject]]` ссылается на `sayHiMixin`, как показано на картинке выше.

Так как `super` ищет родительские методы в `[[HomeObject]].[[Prototype]]`, это означает `sayHiMixin.[[Prototype]]`, а не `User.[[Prototype]]`.

EventMixin

Многие объекты в браузерной разработке (и не только) обладают важной способностью – они могут генерировать события. События – отличный способ передачи информации всем, кто в ней заинтересован. Давайте создадим примесь, которая позволит легко добавлять функциональность по работе с событиями любым классам/объектам.

> Примесь добавит метод `.trigger(name, [data])` для генерации события. Аргумент `name` – это имя

события, за которым могут следовать другие аргументы с данными для события.

Также будет добавлен метод `.on(name, handler)`, который назначает обработчик для события с заданным именем. Обработчик будет вызван, когда произойдёт событие с указанным именем `name`, и получит данные из `.trigger`.

...и метод `.off(name, handler)`, который удаляет обработчик указанного события.

После того, как все методы примеси будут добавлены, объект `user` сможет сгенерировать событие `"login"` после входа пользователя в личный кабинет. А другой объект, к примеру, `calendar` сможет использовать это событие, чтобы показывать зашедшему пользователю актуальный для него календарь.

Или `menu` может генерировать событие `"select"`, когда элемент меню выбран, а другие объекты могут назначать обработчики, чтобы реагировать на это событие, и т.п.

Вот код примеси:

```
let eventMixin = {
  /**
   * Подписаться на событие, использование:
   * menu.on('select', function(item) { ... }
   */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },

  /**
   * Отменить подписку, использование:
   * menu.off('select', handler)
   */
  off(eventName, handler) {
    let handlers = this._eventHandlers &&
this._eventHandlers[eventName];
```

```
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
  },

  /**
   * Сгенерировать событие с указанным именем и данными
   * this.trigger('select', data1, data2);
   */
  trigger(eventName, ...args) {
    if (!this._eventHandlers || !
this._eventHandlers[eventName]) {
      return; // обработчиков для этого события нет
    }

    // вызовем обработчики
    this._eventHandlers[eventName].forEach(handler =>
handler.apply(this, args));
  }
};
```

Итак, у нас есть 3 метода:

1. `.on(eventName, handler)` – назначает функцию `handler`, чтобы обработать событие с заданным именем. Обработчики хранятся в свойстве `_eventHandlers`, представляющим собой объект, в котором имя события является ключом, а массив обработчиков – значением.

2. `.off(eventName, handler)` – убирает функцию из списка обработчиков.

3. `.trigger(eventName, ...args)` – генерирует событие: все назначенные обработчики из `_eventHandlers[eventName]` вызываются, и `...args` передаются им в качестве аргументов.

Использование:

```
// Создадим класс
class Menu {
```

```
  choose(value) {
    this.trigger("select", value);
  }
}
// Добавим примесь с методами для событий
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// Добавить обработчик, который будет вызван при событии
"select":


// Генерирует событие => обработчик выше запускается и
выводит:
menu.choose("123"); // Выбранное значение: 123
menu.on("select", value => alert(`Выбранное
значение: ${value}`));
```

Теперь если у нас есть код, заинтересованный в событии `"select"`, то он может слушать его с помощью `menu.on(...)`.

А `eventMixin` позволяет легко добавить такое поведение в любой класс без вмешательства в цепочку наследования.

Итого

Примесь – общий термин в объектно-ориентированном программировании: класс, который содержит в себе методы для других классов.

Некоторые другие языки допускают множественное наследование. JavaScript не поддерживает множественное наследование, но с помощью примесей мы можем реализовать нечто похожее, скопировав методы в прототип.

Мы можем использовать примеси для расширения функциональности классов, например, для обработки событий, как мы сделали это выше.

С примесями могут возникнуть конфликты, если они перезаписывают существующие методы класса. Стоит помнить об этом и быть внимательнее при выборе имён для методов примеси, чтобы их избежать.

Обработка ошибок

Обработка ошибок, "try..catch"

Неважно, насколько мы хороши в программировании, иногда наши скрипты содержат ошибки. Они могут возникать из-за наших промахов, неожиданного ввода пользователя, неправильного ответа сервера и по тысяче других причин.

Обычно скрипт в случае ошибки «падает» (сразу же останавливается), с выводом ошибки в консоль.

Но есть синтаксическая конструкция `try..catch`, которая позволяет «ловить» ошибки и вместо падения делать что-то более осмысленное.

Синтаксис «try…catch»

Конструкция `try..catch` состоит из двух основных блоков: `try`, и затем `catch`:

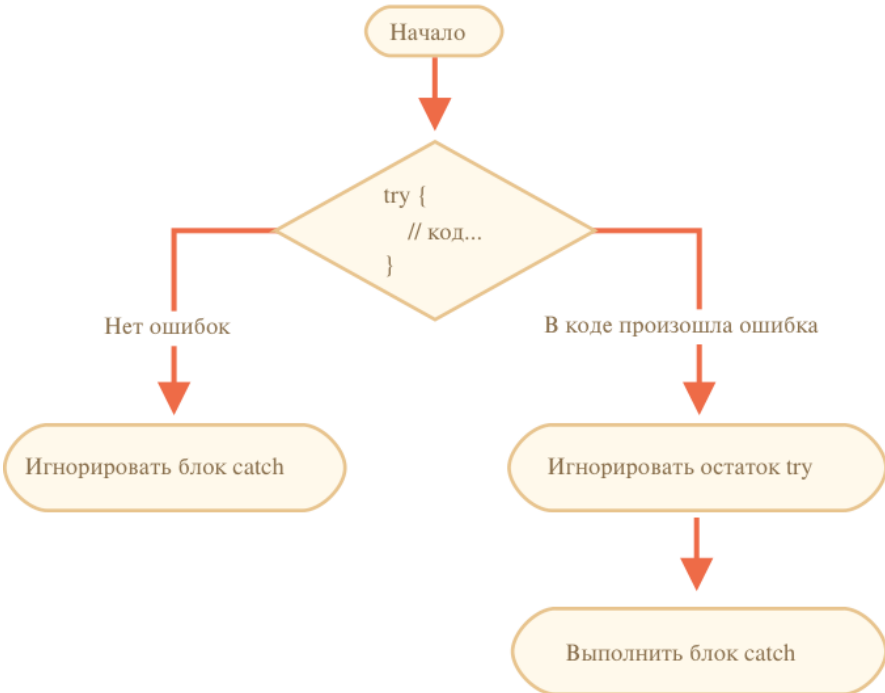
```
try {

  // код...

} catch (err) {

  // обработка ошибки

}
```

Работает она так:

1. Сначала выполняется код внутри блока `try {...}`.
2. Если в нём нет ошибок, то блок `catch(err)` игнорируется: выполнение доходит до конца `try` и

потом далее, полностью пропуская `catch`.

3. Если же в нём возникает ошибка, то выполнение `try` прерывается, и поток управления переходит в начало `catch(err)`. Переменная `err` (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.



Таким образом, при ошибке в блоке `try {…}` скрипт не «падает», и мы получаем возможность обработать ошибку внутри `catch`.

Давайте рассмотрим примеры.

Пример без ошибок: выведет `alert (1)` и `(2)`:

```
try {

  alert('Начало блока try');  //

  // ...код без ошибок
```

```
  alert('Конец блока try');   //

} catch(err) {

  alert('Catch игнорируется, так как нет ошибок'); // (3)

}
(1) <--
(2) <--
```

Пример с ошибками: выведет `(1)` и `(3)`:

```
try {

  alert('Начало блока try');  //


  alert('Конец блока try (никогда не выполнится)');  //
(2)

} catch(err) {

  alert(`Возникла ошибка!`); //

}
(1) <--
  lalala; // ошибка, переменная не
определена!
(3) <--
```

`try..catch` работает только для ошибок, возникающих во время выполнения кода

Чтобы `try..catch` работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код.

Он не сработает, если код синтаксически неверен, например, содержит несовпадающее

количество фигурных скобок:

```
try {
  {{{{{{{{{{{{
} catch(e) {
  alert("Движок не может понять этот код, он
некорректен");
}
```

JavaScript-движок сначала читает код, а затем исполняет его. Ошибки, которые возникают во время фазы чтения, называются ошибками парсинга. Их нельзя обработать (изнутри этого кода), потому что движок не понимает код.

Таким образом, `try..catch` может обрабатывать только ошибки, которые возникают в корректном коде. Такие ошибки называют «ошибками во время выполнения», а иногда «исключениями».

`try..catch` работает синхронно

Исключение, которое произойдёт в коде, запланированном «на будущее», например в `setTimeout`, `try..catch` не поймает:

```
try {
  setTimeout(function() {
    noSuchVariable; // скрипт упадёт тут
  }, 1000);
} catch (e) {
  alert( "не сработает" );
}
```

Это потому, что функция выполняется позже, когда движок уже покинул конструкцию `try..catch`.

Чтобы поймать исключение внутри запланированной функции, `try..catch` должен находиться внутри самой этой функции:

```
setTimeout(function() {
  try {
    noSuchVariable; // try..catch обрабатывает ошибку!
  } catch {
    alert( "ошибка поймана!" );
  }
}, 1000);
```

Объект ошибки

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок `catch`:

```
try {
  // ...
} catch(err) { // <-- объект ошибки, можно использовать другое название вместо err
  // ...
}
```

Для всех встроенных ошибок этот объект имеет два основных свойства:

`name`

Имя ошибки. Например, для неопределённой переменной это `"ReferenceError"`.

`message`

Текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко

используемых и поддерживаемых – это:

`stack`

Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

Например:

```
try {

} catch(err) {
  alert(err.name); // ReferenceError
  alert(err.message); // lalala is not defined
  alert(err.stack); // ReferenceError: lalala is not defined
at (...стек вызовов)

  // Можем также просто вывести ошибку целиком
  // Ошибка приводится к строке вида "name: message"
  alert(err); // ReferenceError: lalala is not defined
}
  lalala; // ошибка, переменная не определена!
```

Блок «catch» без переменной

> **Новая возможность**
>
> Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Если нам не нужны детали ошибки, в `catch` можно её пропустить:

```
try {
  // ...
} catch { //  <-- без (err)
```

```
  // ...
}
```

Использование «try…catch»

Давайте рассмотрим реальные случаи использования `try..catch`.

Как мы уже знаем, JavaScript поддерживает метод JSON.parse(str) для чтения JSON.

Обычно он используется для декодирования данных, полученных по сети, от сервера или из другого источника.

Мы получаем их и вызываем `JSON.parse` вот так:

```
let json = '{"name":"John", "age": 30}'; // данные с сервера


// теперь user – объект со свойствами из строки
alert( user.name ); // John
alert( user.age );  // 30
let user = JSON.parse(json); // преобразовали текстовое представление в JS-объект
```

Вы можете найти более детальную информацию о JSON в главе Формат JSON, метод toJSON.

Если `json` некорректен, `JSON.parse` генерирует ошибку, то есть скрипт «падает».

Устроит ли нас такое поведение? Конечно нет!

Получается, что если вдруг что-то не так с данными, то посетитель никогда (если, конечно, не откроет консоль) об этом не узнает. А люди очень не любят, когда что-то «просто падает» без всякого сообщения об ошибке.

Давайте используем `try..catch` для обработки ошибки:

```
let json = "{ некорректный JSON }";

try {


  alert( user.name ); // не сработает

} catch (e) {

}
  let user = JSON.parse(json); // <-- тут
возникает ошибка...
  // ...выполнение прыгает сюда
  alert( "Извините, в данных ошибка, мы
попробуем получить их ещё раз." );
  alert( e.name );
  alert( e.message );
```

Здесь мы используем блок `catch` только для вывода сообщения, но мы также можем сделать гораздо больше: отправить новый сетевой запрос, предложить посетителю альтернативный способ, отослать информацию об ошибке на сервер для логирования, … Всё лучше, чем просто «падение».

Генерация собственных ошибок

Что если `json` синтаксически корректен, но не содержит необходимого свойства `name`?

Например, так:

```
let json = '{ "age": 30 }'; // данные неполны

try {

  let user = JSON.parse(json); // <-- выполнится без ошибок


} catch (e) {
  alert( "не выполнится" );
}
```

```
alert( user.name ); // нет свойства name!
```

Здесь `JSON.parse` выполнится без ошибок, но на самом деле отсутствие свойства `name` для нас ошибка.

Для того, чтобы унифицировать обработку ошибок, мы воспользуемся оператором `throw`.

Оператор «throw»

Оператор `throw` генерирует ошибку.

Синтаксис:

```
throw <объект ошибки>
```

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами `name` и `message` (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` и другие. Можно использовать и их для создания объектов ошибки.

Их синтаксис:

```
let error = new Error(message);
// или
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство `name` – это в точности имя конструктора. А свойство `message` берётся из аргумента.

Например:

```
let error = new Error(" Ого, ошибка! o_O");

alert(error.name); // Error
alert(error.message); //  Ого, ошибка! o_O
```

Давайте посмотрим, какую ошибку генерирует `JSON.parse`:

```
try {
  JSON.parse("{ bad json o_O }");
} catch(e) {

  alert(e.message); // Unexpected token b in JSON at position
2
}
   alert(e.name); // SyntaxError
```

Как мы видим, это `SyntaxError`.

В нашем случае отсутствие свойства `name` – это ошибка, ведь пользователи должны иметь имена.

Сгенерируем её:

```
let json = '{ "age": 30 }'; // данные неполны

try {

  let user = JSON.parse(json); // <-- выполнится без ошибок

  if (!user.name) {

  }

  alert( user.name );

} catch(e) {
  alert( "JSON Error: " + e.message ); // JSON Error: Данные
неполны: нет имени
}
    throw new SyntaxError("Данные неполны: нет
имени"); // (*)
```

В строке `(*)` оператор `throw` генерирует ошибку

`SyntaxError` с сообщением `message`. Точно такого же вида, как генерирует сам JavaScript. Выполнение блока `try` немедленно останавливается, и поток управления прыгает в `catch`.

Теперь блок `catch` становится единственным местом для обработки всех ошибок: и для `JSON.parse` и для других случаев.

Проброс исключения

В примере выше мы использовали `try..catch` для обработки некорректных данных. А что, если в блоке `try {...}` возникнет *другая неожиданная ошибка*? Например, программная (неопределённая переменная) или какая-то ещё, а не ошибка, связанная с некорректными данными.

Пример:

```
let json = '{ "age": 30 }'; // данные неполны

try {
  user = JSON.parse(json); // <-- забыл добавить "let" перед
user

  // ...
} catch(err) {
  alert("JSON Error: " + err); // JSON Error: ReferenceError:
user is not defined
  // (не JSON ошибка на самом деле)
}
```

Конечно, возможно все! Программисты совершают ошибки. Даже в утилитах с открытым исходным кодом, используемых миллионами людей на протяжении десятилетий – вдруг может быть обнаружена ошибка, которая приводит к ужасным взломам.

В нашем случае `try..catch` предназначен для выявления ошибок, связанных с некорректными данными.

Но по своей природе `catch` получает *все* свои ошибки из `try`. Здесь он получает неожиданную ошибку, но всё также показывает то же самое сообщение `"JSON Error"`. Это неправильно и затрудняет отладку кода.

К счастью, мы можем выяснить, какую ошибку мы получили, например, по её свойству `name`:

```
try {
  user = { /*...*/ };
} catch(e) {

}
  alert(e.name); // "ReferenceError" из-за неопределённой переменной
```

Есть простое правило:

Блок `catch` должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.

Техника «проброс исключения» выглядит так:

1. Блок `catch` получает все ошибки.
2. В блоке `catch(err) {...}` мы анализируем объект ошибки `err`.
3. Если мы не знаем как её обработать, тогда делаем `throw err`.

В коде ниже мы используем проброс исключения, `catch` обрабатывает только `SyntaxError`:

```
let json = '{ "age": 30 }'; // данные неполны
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Данные неполны: нет имени");
  }
```

```
  alert( user.name );

} catch(e) {



}
```

```
  blabla(); // неожиданная ошибка
  if (e.name == "SyntaxError") {
    alert( "JSON Error: " + e.message );
  } else {
    throw e; // проброс (*)
  }
```

Ошибка в строке `(*)` из блока `catch` «выпадает наружу» и может быть поймана другой внешней конструкцией `try..catch` (если есть), или «убьёт» скрипт.

Таким образом, блок `catch` фактически обрабатывает только те ошибки, с которыми он знает, как справляться, и пропускает остальные.

Пример ниже демонстрирует, как такие ошибки могут быть пойманы с помощью ещё одного уровня `try..catch`:

```
function readData() {
  let json = '{ "age": 30 }';

  try {
    // ...

  } catch (e) {
    // ...
    if (e.name != 'SyntaxError') {

    }
  }
}

try {
  readData();
} catch (e) {
```

```
}
    blabla(); // ошибка!
      throw e; // проброс исключения (не знаю
как это обработать)
  alert( "Внешний catch поймал: " + e ); //
поймал!
```

Здесь `readData` знает только, как обработать `SyntaxError`, тогда как внешний блок `try..catch` знает, как обработать всё.

try…catch…finally

Подождите, это ещё не всё.

Конструкция `try..catch` может содержать ещё одну секцию: `finally`.

Если секция есть, то она выполняется в любом случае:

- после `try`, если не было ошибок,
- после `catch`, если ошибки были.

Расширенный синтаксис выглядит следующим образом:

```
try {
   ... пробуем выполнить код...
} catch(e) {
   ... обрабатываем ошибки ...
} finally {
   ... выполняем всегда ...
}
```

Попробуйте запустить такой код:

```
try {
  alert( 'try' );
  if (confirm('Сгенерировать ошибку?')) BAD_CODE();
} catch (e) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

У кода есть два пути выполнения:

1. Если вы ответите на вопрос «Сгенерировать ошибку?» утвердительно, то `try -> catch -> finally`.
2. Если ответите отрицательно, то `try -> finally`.

Секцию `finally` часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от того, будет ошибка или нет.

Например, мы хотим измерить время, которое занимает функция чисел Фибоначчи `fib(n)`. Естественно, мы можем начать измерения до того, как функция начнёт выполняться и закончить после. Но что делать, если при вызове функции возникла ошибка? В частности, реализация `fib(n)` в коде ниже возвращает ошибку для отрицательных и для нецелых чисел.

Секция `finally` отлично подходит для завершения измерений несмотря ни на что.

Здесь `finally` гарантирует, что время будет измерено корректно в обеих ситуациях – и в случае успешного завершения `fib` и в случае ошибки:

```
let num = +prompt("Введите положительное целое число?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Должно быть целое неотрицательное
число");
  }
```

```
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

let start = Date.now();

try {
  result = fib(num);
} catch (e) {
  result = 0;


alert(result || "возникла ошибка");

alert( `Выполнение заняло ${diff}ms` );
} finally {
  diff = Date.now() - start;
}
```

Вы можете это проверить, запустив этот код и введя `35` в `prompt` – код завершится нормально, `finally` выполнится после `try`. А затем введите `-1` – незамедлительно произойдёт ошибка, выполнение займёт `0ms`. Оба измерения выполняются корректно.

Другими словами, неважно как завершилась функция: через `return` или `throw`. Секция `finally` срабатывает в обоих случаях.

Переменные внутри `try..catch..finally` локальны

Обратите внимание, что переменные `result` и `diff` в коде выше объявлены *до* `try..catch`.

Если переменную объявить в блоке, например, в `try`, то она не будет доступна после него.

`finally` и `return`

Блок `finally` срабатывает при *любом* выходе из `try..catch`, в том числе и `return`.

В примере ниже из `try` происходит `return`, но `finally` получает управление до того, как контроль возвращается во внешний код.

```
function func() {

  try {


  } catch (e) {
    /* ... */
  } finally {

  }
}

alert( func() ); // сначала срабатывает alert из finally, а затем этот код
    return 1;
    alert( 'finally' );
```

try..finally

Конструкция `try..finally` без секции `catch` также полезна. Мы применяем её, когда не хотим здесь обрабатывать ошибки (пусть выпадут), но хотим быть уверены, что начатые процессы завершились.

```
function func() {
  // начать делать что-то, что требует завершения (например, измерения)
  try {
    // ...
  } finally {
    // завершить это, даже если все упадёт
  }
}
```

В приведённом выше коде ошибка всегда выпадает наружу, потому что тут нет блока `catch`. Но `finally` отрабатывает до того, как поток управления выйдет из функции.

Глобальный catch

Зависит от окружения

Информация из данной секции не является частью языка JavaScript.

Давайте представим, что произошла фатальная ошибка (программная или что-то ещё ужасное) снаружи `try..catch`, и скрипт упал.

Существует ли способ отреагировать на такие ситуации? Мы можем захотеть залогировать ошибку, показать что-то пользователю (обычно они не видят сообщение об ошибке) и т.д.

Такого способа нет в спецификации, но обычно окружения предоставляют его, потому что это весьма полезно. Например, в Node.js для этого есть `process.on("uncaughtException")`. А в браузере мы можем присвоить функцию специальному свойству window.onerror, которая будет вызвана в случае необработанной ошибки.

Синтаксис:

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

message

Сообщение об ошибке.

url

URL скрипта, в котором произошла ошибка.

line, col

Номера строки и столбца, в которых произошла ошибка.

error

Объект ошибки.

Пример:

```
<script>


  function readData() {
    badFunc(); // Ой, что-то пошло не так!
  }

  readData();
</script>
  window.onerror = function(message, url,
line, col, error) {
    alert(`${message}\n В ${line}:${col} на
${url}`);
  };
```

Роль глобального обработчика `window.onerror` обычно заключается не в восстановлении выполнения скрипта – это скорее всего невозможно в случае программной ошибки, а в отправке сообщения об ошибке разработчикам.

Существуют также веб-сервисы, которые предоставляют логирование ошибок для таких случаев, такие как https://

errorception.com или http://www.muscula.com.

Они работают так:

1. Мы регистрируемся в сервисе и получаем небольшой JS-скрипт (или URL скрипта) от них для вставки на страницы.
2. Этот JS-скрипт ставит свою функцию `window.onerror`.
3. Когда возникает ошибка, она выполняется и отправляет сетевой запрос с информацией о ней в сервис.
4. Мы можем войти в веб-интерфейс сервиса и увидеть ошибки.

Итого

Конструкция `try..catch` позволяет обрабатывать ошибки во время исполнения кода. Она позволяет запустить код и перехватить ошибки, которые могут в нём возникнуть.

Синтаксис:

```
try {
  // исполняем код
} catch(err) {
  // если случилась ошибка, прыгаем сюда
  // err – это объект ошибки
} finally {
  // выполняется всегда после try/catch
}
```

Секций `catch` или `finally` может не быть, то есть более короткие конструкции `try..catch` и `try..finally` также корректны.

Объекты ошибок содержат следующие свойства:

`message` – понятное человеку сообщение.

`name` – строка с именем ошибки (имя конструктора ошибки).

`stack` (нестандартное, но хорошо поддерживается) – стек на момент ошибки.

Если объект ошибки не нужен, мы можем пропустить его, используя `catch {` вместо `catch(err) {`.

Мы можем также генерировать собственные ошибки, используя оператор `throw`. Аргументом `throw` может быть что угодно, но обычно это объект ошибки, наследуемый от встроенного класса `Error`. Подробнее о расширении ошибок см. в следующей главе.

Проброс исключения – это очень важный приём обработки ошибок: блок `catch` обычно ожидает и знает, как обработать определённый тип ошибок, поэтому он должен пробрасывать дальше ошибки, о которых он не знает.

Даже если у нас нет `try..catch`, большинство сред позволяют настроить «глобальный» обработчик ошибок, чтобы ловить ошибки, которые «выпадают наружу». В браузере это `window.onerror`.

Задачи

Finally или просто код?

важность: 5

Сравните два фрагмента кода.

1. Первый использует `finally` для выполнения кода после `try..catch`:

```
try {
  начать работу
  работать
} catch (e) {
  обработать ошибку
} finally {

}
   очистить рабочее пространство
```

2. Второй фрагмент просто ставит очистку после `try..catch`:

```
try {
  начать работу
  работать
} catch (e) {
  обработать ошибку
}

очистить рабочее пространство
```

Нам определённо нужна очистка после работы, неважно возникли ошибки или нет.

Есть ли здесь преимущество в использовании `finally` или оба фрагмента кода одинаковы? Если такое преимущество есть, то дайте пример, когда оно проявляется.

К решению

Пользовательские ошибки, расширение Error

Когда что-то разрабатываем, то нам часто необходимы собственные классы ошибок для разных вещей, которые

могут пойти не так в наших задачах. Для ошибок при работе с сетью может понадобиться `HttpError`, для операций с базой данных `DbError`, для поиска – `NotFoundError` и т.д.

Наши ошибки должны поддерживать базовые свойства, такие как `message`, `name` и, желательно, `stack`. Но также они могут иметь свои собственные свойства. Например, объекты `HttpError` могут иметь свойство `statusCode` со значениями `404`, `403` или `500`.

JavaScript позволяет вызывать `throw` с любыми аргументами, то есть технически наши классы ошибок не нуждаются в наследовании от `Error`. Но если использовать наследование, то появляется возможность идентификации объектов ошибок посредством `obj instanceof Error`. Так что лучше применять наследование.

По мере роста приложения, наши собственные ошибки образуют иерархию, например, `HttpTimeoutError` может наследовать от `HttpError` и так далее.

Расширение Error

В качестве примера рассмотрим функцию `readUser(json)`, которая должна читать данные пользователя в формате JSON.

Пример того, как может выглядеть корректный `json`:

```
let json = `{ "name": "John", "age": 30 }`;
```

Внутри будем использовать `JSON.parse`. При получении некорректного `json` он будет генерировать ошибку `SyntaxError`. Но даже если `json` синтаксически верен, то это не значит, что это будет корректный пользователь, верно? Могут быть пропущены необходимые данные.

Например, могут отсутствовать свойства `name`и `age`, которые являются необходимыми для наших пользователей.

Наша функция `readUser(json)` будет не только читать JSON-данные, но и проверять их («валидировать»). Если необходимые поля отсутствуют или данные в неверном формате, то это будет ошибкой. Но не синтаксической ошибкой `SyntaxError`, потому что данные синтаксически корректны. Это будет другая ошибка.

Назовём её ошибкой валидации `ValidationError` и создадим для неё класс. Ошибка этого вида должна содержать информацию о поле, которое является источником ошибки.

Наш класс `ValidationError` должен наследовать от встроенного класса `Error`.

Класс `Error` встроенный, вот его примерный код, просто чтобы мы понимали, что расширяем:

```
// "Псевдокод" встроенного класса Error, определённого самим JavaScript
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (разные имена для разных встроенных классов ошибок)
    this.stack = <стек вызовов>; // нестандартное свойство, но обычно поддерживается
  }
}
```

Теперь давайте унаследуем от него `ValidationError` и попробуем новый класс в действии:

```
  constructor(message) {
    super(message); // (1)
    this.name = "ValidationError"; // (2)
  }
}
```

```
function test() {
  throw new ValidationError("Упс!");
}

try {
  test();
} catch(err) {
  alert(err.message); // Упс!
  alert(err.name); // ValidationError
  alert(err.stack); // список вложенных вызовов с номерами
строк для каждого
}
class ValidationError extends Error {
```

Обратите внимание: в строке `(1)` вызываем родительский конструктор. JavaScript требует от нас вызова `super` в дочернем конструкторе, так что это обязательно. Родительский конструктор устанавливает свойство `message`.

Родительский конструктор также устанавливает свойство `name` для `"Error"`, поэтому в строке `(2)` мы сбрасываем его на правильное значение.

Попробуем использовать его в `readUser(json)`:

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

// Использование
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new ValidationError("Нет поля: age");
  }
  if (!user.name) {
    throw new ValidationError("Нет поля: name");
  }

  return user;
}
```

```
// Рабочий пример с try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {

  } else if (err instanceof SyntaxError) { // (*)
    alert("JSON Ошибка Синтаксиса: " + err.message);
  } else {
    throw err; // неизвестная ошибка, пробросить исключение
(**)
  }
}
```

```
    alert("Некорректные данные: " +
err.message); // Некорректные данные: Нет
поля: name
```

Блок `try..catch` в коде выше обрабатывает и нашу `ValidationError`, и встроенную `SyntaxError` из `JSON.parse`.

Обратите внимание, как мы используем `instanceof` для проверки конкретного типа ошибки в строке `(*)`.

Мы можем также проверить тип, используя `err.name`:

```
// ...
// вместо (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

Версия с `instanceof` гораздо лучше, потому что в будущем мы собираемся расширить `ValidationError`, сделав его подтипы, такие как `PropertyRequiredError`. И проверка `instanceof` продолжит работать для новых наследованных классов. Так что это на будущее.

Также важно, что если `catch` встречает неизвестную ошибку, то он пробрасывает её в строке `(**)`. Блок`catch` знает, только как обрабатывать ошибки валидации и синтаксические ошибки, а другие виды ошибок (из-за

опечаток в коде и другие непонятные) он должен выпустить наружу.

Дальнейшее наследование

Класс `ValidationError` является слишком общим. Много что может пойти не так. Свойство может отсутствовать или иметь неверный формат (например, строка как значение возраста `age`). Поэтому для отсутствующих свойств сделаем более конкретный класс `PropertyRequiredError`. Он будет нести дополнительную информацию о свойстве, которое отсутствует.

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}



// Применение
function readUser(json) {
  let user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError("age");
  }
  if (!user.name) {
    throw new PropertyRequiredError("name");
  }

  return user;
}

// Рабочий пример с try..catch

try {
  let user = readUser('{ "age": 25 }');
} catch (err) {
  if (err instanceof ValidationError) {

  } else if (err instanceof SyntaxError) {
```

```
    alert("Ошибка синтаксиса JSON: " + err.message);
  } else {
    throw err; // неизвестная ошибка, повторно выбросит
исключение
  }
}
class PropertyRequiredError extends
ValidationError {
  constructor(property) {
    super("Нет свойства: " + property);
    this.name = "PropertyRequiredError";
    this.property = property;
  }
}
    alert("Неверные данные: " + err.message);
// Неверные данные: Нет свойства: name
    alert(err.name); // PropertyRequiredError
    alert(err.property); // name
```

Новый класс `PropertyRequiredError` очень просто использовать: необходимо указать только имя свойства `new PropertyRequiredError(property)`. Сообщение для пользователя `message` генерируется конструктором.

Обратите внимание, что свойство `this.name` в конструкторе `PropertyRequiredError` снова присвоено вручную. Правда, немного утомительно – присваивать `this.name = <class name>` в каждом классе пользовательской ошибки. Можно этого избежать, если сделать наш собственный «базовый» класс ошибки, который будет ставить `this.name = this.constructor.name`. И затем наследовать все ошибки уже от него.

Давайте назовём его `MyError`.

Вот упрощённый код с `MyError` и другими пользовательскими классами ошибок:

```
class MyError extends Error {
  constructor(message) {
```

```
    super(message);

  }
}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super("Нет свойства: " + property);
    this.property = property;
  }
}

// name корректное
alert( new PropertyRequiredError("field").name ); //
PropertyRequiredError
    this.name = this.constructor.name;
```

Теперь пользовательские ошибки стали намного короче, особенно `ValidationError`, так как мы избавились от строки `"this.name = ..."` в конструкторе.

Обёртывание исключений

Назначение функции `readUser` в приведённом выше коде – это «чтение данных пользователя». В процессе могут возникнуть различные виды ошибок. Сейчас у нас есть `SyntaxError` и `ValidationError`, но в будущем функция `readUser` может расшириться и, возможно, генерировать другие виды ошибок.

Код, который вызывает `readUser`, должен обрабатывать эти ошибки.

Сейчас в нём используются проверки `if` в блоке `catch`, которые проверяют класс и обрабатывают известные ошибки и пробрасывают дальше неизвестные. Но если функция `readUser` генерирует несколько видов ошибок, то мы должны спросить себя: действительно ли мы хотим проверять все типы ошибок поодиночке во всех местах в коде, где вызывается `readUser`?

Часто ответ «Нет»: внешний код хочет быть на один уровень выше всего этого. Он хочет иметь какую-то обобщённую ошибку чтения данных. Почему именно это произошло – часто не имеет значения (об этом говорится в сообщении об ошибке). Или даже лучше, если есть способ получить подробности об ошибке, но только если нам это нужно.

Итак, давайте создадим новый класс `ReadError` для представления таких ошибок. Если ошибка возникает внутри `readUser`, мы её перехватим и сгенерируем `ReadError`. Мы также сохраним ссылку на исходную ошибку в свойстве `cause`. Тогда внешний код должен будет только проверить наличие `ReadError`.

Этот код определяет ошибку `ReadError` и демонстрирует её использование в `readUser`и `try..catch`:

```
class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ...
*/ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
```

```
  }

  try {
    validateUser(user);
  } catch (err) {

  }

}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {

  } else {
    throw e;
  }
}
```

```
    if (err instanceof SyntaxError) {
      throw new ReadError("Синтаксическая
ошибка", err);
    } else {
      throw err;
    }
    if (err instanceof ValidationError) {
      throw new ReadError("Ошибка валидации",
err);
    } else {
      throw err;
    }
    alert(e);
    // Исходная ошибка: SyntaxError:Unexpected
token b in JSON at position 1
    alert("Исходная ошибка: " + e.cause);
```

В приведённом выше коде `readUser` работает так, как описано – функция распознаёт синтаксические ошибки и ошибки валидации и выдаёт вместо них ошибки `ReadError` (неизвестные ошибки, как обычно, пробрасываются).

Внешний код проверяет только `instanceof ReadError`.

Не нужно перечислять все возможные типы ошибок

Этот подход называется «обёртывание исключений», потому что мы берём «исключения низкого уровня» и «оборачиваем» их в `ReadError`, который является более абстрактным и более удобным для использования в вызывающем коде. Такой подход широко используется в объектно-ориентированном программировании.

Итого

- Мы можем наследовать свои классы ошибок от `Error` и других встроенных классов ошибок, но нужно позаботиться о свойстве `name` и не забыть вызвать `super`.
- Мы можем использовать `instanceof` для проверки типа ошибок. Это также работает с наследованием. Но иногда у нас объект ошибки, возникшей в сторонней библиотеке, и нет простого способа получить класс. Тогда для проверки типа ошибки можно использовать свойство `name`.
- Обёртывание исключений является распространённой техникой: функция ловит низкоуровневые исключения и создаёт одно «высокоуровневое» исключение вместо разных низкоуровневых. Иногда низкоуровневые исключения становятся свойствами этого объекта, как `err.cause` в примерах выше, но это не обязательно.

Задачи

Наследование от SyntaxError

важность: 5

Создайте класс `FormatError`, который наследует от

встроенного класса `SyntaxError`.

Класс должен поддерживать свойства `message`, `name` и `stack`.

Пример использования:

```
let err = new FormatError("ошибка форматирования");

alert( err.message ); // ошибка форматирования
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof FormatError ); // true
alert( err instanceof SyntaxError ); // true (потому что
наследует от SyntaxError)
```

К решению

Промисы, async/await

Введение: колбэки

Многие действия в JavaScript *асинхронные*.

Например, рассмотрим функцию `loadScript(src)`:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Эта функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция `<script src="…">`, браузер загрузит скрипт и выполнит его.

Вот пример использования этой функции:

```
// загрузит и выполнит скрипт
loadScript('/my/script.js');
```

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Если после вызова `loadScript(…)` есть какой-то код, то он не будет ждать, пока скрипт загрузится.

```
loadScript('/my/script.js');
// код, написанный после вызова функции loadScript,
// не будет дожидаться полной загрузки скрипта
// ...
```

Мы хотели бы использовать новый скрипт, как только он будет загружен. Скажем, он объявляет новую функцию,

которую мы хотим выполнить.

Но если мы просто вызовем эту функцию после `loadScript(…)`, у нас ничего не выйдет:

```
loadScript('/my/script.js'); // в скрипте есть "function
newFunction() {…}"

newFunction(); // такой функции не существует!
```

Действительно, ведь у браузера не было времени загрузить скрипт. Сейчас функция `loadScript` никак не позволяет отследить момент загрузки. Скрипт загружается, а потом выполняется. Но нам нужно точно знать, когда это произойдёт, чтобы использовать функции и переменные из этого скрипта.

Давайте передадим функцию `callback` вторым аргументом в `loadScript`, чтобы вызвать её, когда скрипт загрузится:

```
function loadScript(src, ) {
  let script = document.createElement('script');
  script.src = src;



  document.head.append(script);
}
callback
  script.onload = () => callback(script);
```

Теперь, если мы хотим вызвать функцию из скрипта, нужно делать это в колбэке:

```
loadScript('/my/script.js', function() {
  // эта функция вызовется после того, когда загрузится скрипт
  newFunction(); // теперь всё работает
  ...
});
```

Смысл такой: вторым аргументом передаётся функция (обычно анонимная), которая выполняется по завершении действия.

Возьмём для примера реальный скрипт с библиотекой функций:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}
```

```
loadScript('https://cdnjs.cloudflare.com/ajax/
libs/lodash.js/3.2.0/lodash.js', script => {
  alert(`Здорово, скрипт ${script.src}
загрузился`);
  alert( _ ); // функция, объявленная в
загруженном скрипте
});
```

Такое написание называют асинхронным программированием с использованием колбэков. В функции, которые выполняют какие-либо асинхронные операции, передаётся аргумент `callback` — функция, которая будет вызвана по завершению асинхронного действия.

Мы поступили похожим образом в `loadScript`, но это, конечно, распространённый подход.

Колбэк в колбэке

Как нам загрузить два скрипта один за другим: сначала первый, а за ним второй?

Первое, что приходит в голову, вызвать `loadScript` ещё раз уже внутри колбэка, вот так:

```
loadScript('/my/script.js', function(script) {

  alert(`Здорово, скрипт ${script.src} загрузился, загрузим ещё один`);



});
  loadScript('/my/script2.js', function(script) {
    alert(`Здорово, второй скрипт загрузился`);
  });
```

Когда внешняя функция `loadScript` выполнится, вызовется та, что внутри колбэка.

А что если нам нужно загрузить ещё один скрипт?..

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {



  })

});
    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут загружены
    });
```

Каждое новое действие мы вынуждены вызывать внутри колбэка. Этот вариант подойдёт, когда у нас одно-два действия, но для большего количества уже не удобно. Альтернативные подходы мы скоро разберём.

Перехват ошибок

В примерах выше мы не думали об ошибках. А что если загрузить скрипт не удалось? Колбэк должен уметь реагировать на возможные проблемы.

Ниже улучшенная версия `loadScript`, которая умеет отслеживать ошибки загрузки:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;



  document.head.append(script);
}
  script.onload = () => callback(null,
script);
  script.onerror = () => callback(new
Error(`Не удалось загрузить скрипт ${src}`));
```

Мы вызываем `callback(null, script)` в случае успешной загрузки и `callback(error)`, если загрузить скрипт не удалось.

Живой пример:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // обрабатываем ошибку
  } else {
    // скрипт успешно загружен
  }
});
```

Опять же, подход, который мы использовали в `loadScript`, также распространён и называется «колбэк с первым аргументом-ошибкой» («error-first callback»).

Правила таковы:

1. Первый аргумент функции `callback` зарезервирован для ошибки. В этом случае вызов выглядит вот так:

`callback(err)`.

2. Второй и последующие аргументы — для результатов выполнения. В этом случае вызов выглядит вот так: `callback(null, result1, result2…)`.

Одна и та же функция `callback` используется и для информирования об ошибке, и для передачи результатов.

Адская пирамида вызовов

На первый взгляд это рабочий способ написания асинхронного кода. Так и есть. Для одного или двух вложенных вызовов всё выглядит нормально.

Но для нескольких асинхронных действий, которые нужно выполнить друг за другом, код выглядит вот так:

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {

          }
        });

      }
    })
  }
});
              // ...и так далее, пока все
скрипты не будут загружены (*)
```

В примере выше:

1. Мы загружаем `1.js`. Продолжаем, если нет ошибок.
2. Мы загружаем `2.js`. Продолжаем, если нет ошибок.
3. Мы загружаем `3.js`. Продолжаем, если нет ошибок. И так далее `(*)`.

Чем больше вложенных вызовов, тем наш код будет иметь всё большую вложенность, которую сложно поддерживать, особенно если вместо `...` у нас код, содержащий другие цепочки вызовов, условия и т.д.

Иногда это называют «адом колбэков» или «адской пирамидой колбэков».

```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...
          }
        });
      }
    })
  }
});
```

Пирамида вложенных вызовов растёт вправо с каждым асинхронным действием. В итоге вы сами будете путаться, где что есть.

Такой подход к написанию кода не приветствуется.

Мы можем попытаться решить эту проблему, изолируя каждое действие в отдельную функцию, вот так:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...и так далее, пока все скрипты не будут загружены (*)
  }
};
```

Заметили? Этот код делает всё то же самое, но вложенность отсутствует, потому что все действия вынесены в отдельные функции.

Код абсолютно рабочий, но кажется разорванным на куски. Его трудно читать, вы наверняка заметили это. Приходится прыгать глазами между кусками кода, когда пытаешься его прочесть. Это неудобно, особенно, если читатель не знаком с кодом и не знает, что за чем следует.

Кроме того, все функции `step*` одноразовые, и созданы лишь только, чтобы избавиться от «адской пирамиды вызовов». Никто не будет их переиспользовать где-либо ещё. Таким образом, мы, кроме всего прочего, засоряем пространство имён.

Нужно найти способ получше.

К счастью, такие способы существуют. Один из лучших — использовать промисы, о которых рассказано в следующей главе.

Задачи

Анимация круга с помощью колбэка

В задаче Анимированный круг находится код для анимации появления круга.

Давайте представим, что теперь нам нужен не просто круг, а круг с сообщением внутри. И сообщение должно появляться *после* анимации (когда круг достигнет своих размеров), иначе это будет некрасиво.

В том решении функция `showCircle(cx, cy, radius)` рисовала круг, но способа узнать, что всё нарисовано, не было.

Поэтому добавим в параметры колбэк: `showCircle(cx, cy, radius, callback)`, который выполним, когда анимация будет завершена. Функция `callback` будет добавлять в наш круг `<div>` элемент с сообщением.

Посмотрите пример:

```
showCircle(150, 150, 100, div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

Демо:

Возьмите за основу решение задачи Анимированный круг.

К решению

Промисы

Представьте, что вы известный певец, которого фанаты постоянно донимают расспросами о предстоящем сингле.

Чтобы получить передышку, вы обещаете разослать им сингл, когда он будет выпущен. Вы даёте фанатам список, в который они могут записаться. Они могут оставить там свой e-mail, чтобы получить песню, как только она выйдет. И даже больше: если что-то пойдёт не так, например, в студии будет пожар и песню выпустить не выйдет, они также получат уведомление об этом.

Все счастливы! Вы счастливы, потому что вас больше не донимают фанаты, а фанаты могут больше не беспокоиться, что пропустят новый сингл.

Это аналогия из реальной жизни для ситуаций, с которыми мы часто сталкиваемся в программировании:

1. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. В нашей аналогии это – «певец».
2. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции. Это – «фанаты».
3. `Promise` (по англ. `promise`, будем называть такой объект «промис») – это специальный объект в

JavaScript, который связывает «создающий» и «потребляющий» коды вместе. В терминах нашей аналогии – это «список для подписки». «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а *промис* делает результат доступным для кода, который подписан на него, когда результат готов.

Аналогия не совсем точна, потому что объект `Promise` в JavaScript гораздо сложнее простого списка подписок: он обладает дополнительными возможностями и ограничениями. Но для начала и такая аналогия хороша.

Синтаксис создания `Promise`:

```
let promise = new Promise(function(resolve, reject) {
  // функция-исполнитель (executor)
  // "певец"
});
```

Функция, переданная в конструкцию `new Promise`, называется *исполнитель* (executor). Когда `Promise` создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: *исполнитель* – это «певец».

Её аргументы `resolve` и `reject` – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

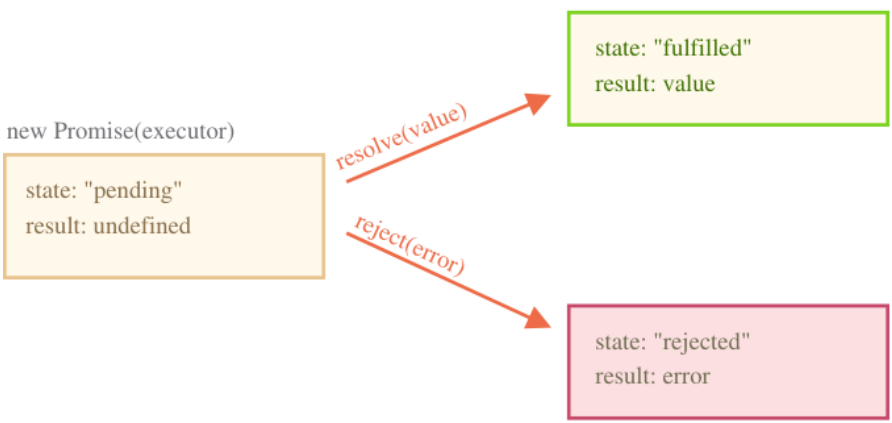
- `resolve(value)` — если работа завершилась успешно, с результатом `value`.
- `reject(error)` — если произошла ошибка, `error` – объект ошибки.

Итак, исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать `resolve` или `reject`.

У объекта `promise`, возвращаемого конструктором `new Promise`, есть внутренние свойства:

- `state` («состояние») — вначале `"pending"` («ожидание»), потом меняется на `"fulfilled"` («выполнено успешно») при вызове `resolve` или на `"rejected"` («выполнено с ошибкой») при вызове `reject`.
- `result` («результат») — вначале `undefined`, далее изменяется на `value` при вызове `resolve(value)` или на `error` при вызове `reject(error)`.

Так что исполнитель по итогу переводит `promise` в одно из двух состояний:



Позже мы рассмотрим, как «фанаты» узнают об этих изменениях.

Ниже пример конструктора `Promise` и простого исполнителя с кодом, дающим результат с задержкой (через `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // эта функция выполнится автоматически, при вызове new
```

```
Promise

  // через 1 секунду сигнализировать, что задача выполнена с
результатом "done"
  setTimeout(() => , 1000);
});
resolve
("done")
```

Мы можем наблюдать две вещи, запустив код выше:

1. Функция-исполнитель запускается сразу же при вызове `new Promise`.
2. Исполнитель получает два аргумента: `resolve` и `reject` — это функции, встроенные в JavaScript, поэтому нам не нужно их писать. Нам нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

Спустя одну секунду «обработки» исполнитель вызовет `resolve("done")`, чтобы передать результат:

Это был пример успешно выполненной задачи, в результате мы получили «успешно выполненный» промис.

А теперь пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:

```
let promise = new Promise(function(resolve, reject) {
  // спустя одну секунду будет сообщено, что задача выполнена
с ошибкой
  setTimeout(() => , 1000);
});
reject
(new Error("Whoops!"))
```

Подведём промежуточные итоги: исполнитель выполняет задачу (что-то, что обычно требует времени), затем вызывает `resolve` или `reject`, чтобы изменить состояние соответствующего `Promise`.

Промис – и успешный, и отклонённый будем называть «завершённым», в отличие от изначального промиса «в ожидании».

Может быть что-то одно: либо результат, либо ошибка

Исполнитель должен вызвать что-то одно: `resolve` или `reject`. Состояние промиса может быть изменено только один раз.

Все последующие вызовы `resolve` и `reject` будут проигнорированы:

```
let promise = new Promise(function(resolve, reject)
{


  reject(new Error("…")); // игнорируется
  setTimeout(() => resolve("…")); // игнорируется
});
   resolve("done");
```

Идея в том, что задача, выполняемая исполнителем, может иметь только один итог: результат или ошибку.

Также заметим, что функция `resolve/reject`

ожидает только один аргумент (или ни одного). Все дополнительные аргументы будут проигнорированы.

Вызывайте `reject` с объектом `Error`

В случае, если что-то пошло не так, мы должны вызвать `reject`. Это можно сделать с аргументом любого типа (как и `resolve`), но рекомендуется использовать объект `Error` (или унаследованный от него). Почему так? Скоро нам станет понятно.

Вызов `resolve`/`reject` сразу

Обычно исполнитель делает что-то асинхронное и после этого вызывает `resolve`/`reject`, то есть через какое-то время. Но это не обязательно, `resolve` или `reject` могут быть вызваны сразу:

```
let promise = new Promise(function(resolve, reject)
{
  // задача, не требующая времени
  resolve(123); // мгновенно выдаст результат: 123
});
```

Это может случиться, например, когда мы начали выполнять какую-то задачу, но тут же увидели, что ранее её уже выполняли, и результат закеширован.

Такая ситуация нормальна. Мы сразу получим успешно завершённый `Promise`.

Свойства `state` и `result` – внутренние

Свойства `state` и `result` – это внутренние свойства объекта `Promise` и мы не имеем к ним прямого доступа. Для обработки результата следует использовать методы `.then`/`.catch`/`.finally`, про них речь пойдёт дальше.

Потребители: then, catch, finally

Объект `Promise` служит связующим звеном между исполнителем («создающим» кодом или «певцом») и функциями-потребителями («фанатами»), которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов `.then`, `.catch` и `.finally`.

then

Наиболее важный и фундаментальный метод – `.then`.

Синтаксис:

```
promise.then(
  function(result) {  },
  function(error) {  }
);
/* обработает успешное выполнение */
/* обработает ошибку */
```

Первый аргумент метода `.then` – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент `.then` – функция, которая выполняется, когда промис переходит в состояние «выполнен с

ошибкой», и получает ошибку.

Например, вот реакция на успешно выполненный промис:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve запустит первую функцию, переданную в .then
promise.then(

  error => alert(error) // не будет запущена
);
   result => alert(result), // выведет "done!"
через одну секунду
```

Выполнилась первая функция.

А в случае ошибки в промисе – выполнится вторая:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject запустит вторую функцию, переданную в .then
promise.then(
  result => alert(result), // не будет запущена

);
   error => alert(error) // выведет "Error:
Whoops!" спустя одну секунду
```

Если мы заинтересованы только в результате успешного выполнения задачи, то в `then` можно передать только одну функцию:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // выведет "done!" спустя
одну секунду
```

catch

Если мы хотели бы только обработать ошибку, то можно использовать `null` в качестве первого аргумента: `.then(null, errorHandlingFunction)`. Или можно воспользоваться методом `.catch(errorHandlingFunction)`, который сделает тоже самое:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});

// .catch(f) это тоже самое, что
promise.then(null, f)
promise.catch(alert); // выведет "Error:
Ошибка!" спустя одну секунду
```

Вызов `.catch(f)` – это сокращённый, «укороченный» вариант `.then(null, f)`.

finally

По аналогии с блоком `finally` из обычного `try {...} catch {...}`, у промисов также есть метод `finally`.

Вызов `.finally(f)` похож на `.then(f, f)`, в том смысле, что `f` выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

`finally` хорошо подходит для очистки, например остановки индикатора загрузки, его ведь нужно остановить вне зависимости от результата.

Например:

```
new Promise((resolve, reject) => {
  /* сделать что-то, что займёт время, и после вызвать
resolve/reject */
})

  .then(result => показать результат, err => показать ошибку)
   // выполнится, когда промис завершится,
```

```
независимо от того, успешно или нет
  .finally(() => остановить индикатор
загрузки)
```

Но это не совсем псевдоним `then(f,f)`, как можно было подумать. Существует несколько важных отличий:

1. Обработчик, вызываемый из `finally`, не имеет аргументов. В `finally` мы не знаем, как был завершён промис. И это нормально, потому что обычно наша задача – выполнить «общие» завершающие процедуры.

2. Обработчик `finally` «пропускает» результат или ошибку дальше, к последующим обработчикам.

   Например, здесь результат проходит через `finally` к `then`:

   ```
   new Promise((resolve, reject) => {
     setTimeout(() => resolve("result"), 2000)
   })
     .finally(() => alert("Промис завершён"))
     .then(result => alert(result)); // <-- .then обработает
   результат
   ```

   А здесь ошибка из промиса проходит через `finally` к `catch`:

   ```
   new Promise((resolve, reject) => {
     throw new Error("error");
   })
     .finally(() => alert("Промис завершён"))
     .catch(err => alert(err));  // <-- .catch обработает
   объект ошибки
   ```

   Это очень удобно, потому что `finally` не предназначен для обработки результата промиса. Так что он просто пропускает его через себя дальше.

   Мы более подробно поговорим о создании цепочек

промисов и передаче результатов между обработчиками в следующей главе.

3. Последнее, но не менее значимое: вызов `.finally(f)` удобнее, чем `.then(f, f)` – не надо дублировать функции f.

На завершённых промисах обработчики запускаются сразу

Если промис в состоянии ожидания, обработчики в `.then/catch/finally` будут ждать его. Однако, если промис уже завершён, то обработчики выполнятся сразу:

```
// при создании промиса он сразу переводится в
состояние "успешно завершён"
let promise = new Promise(resolve =>
resolve("готово!"));

promise.then(alert); // готово! (выведется сразу)
```

Теперь рассмотрим несколько практических примеров того, как промисы могут облегчить нам написание асинхронного кода.

Пример: loadScript

У нас есть функция `loadScript` для загрузки скрипта из предыдущей главы.

Давайте вспомним, как выглядел вариант с колбэками:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Ошибка загрузки
скрипта ${src}`));

  document.head.append(script);
}
```

Теперь перепишем её, используя `Promise`.

Новой функции `loadScript` более не нужен аргумент `callback`. Вместо этого она будет создавать и возвращать объект `Promise`, который перейдет в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя `.then`:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки
скрипта ${src}`));

    document.head.append(script);
  });
}
```

Применение:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/
libs/lodash.js/4.17.11/lodash.js");

promise.then(
  script => alert(`${script.src} загружен!`),
  error => alert(`Ошибка: ${error.message}`)
);

promise.then(script => alert('Ещё один обработчик...'));
```

Сразу заметно несколько преимуществ перед подходом с использованием колбэков:

Промисы	**Колбэки**
Промисы позволяют делать вещи в естественном порядке. Сперва мы запускаем `loadScript(script)`, и затем (`.then`) мы пишем, что делать с результатом.	У нас должна быть функция`callback` на момент вызова `loadScript(script, callback)`. Другими словами, нам нужно знать что делать с результатом *до того*, как вызовется `loadScript`.
Мы можем вызывать `.then` у `Promise` столько раз, сколько захотим. Каждый раз мы добавляем нового «фаната», новую функцию-подписчика в «список подписок». Больше об этом в следующей главе: Цепочка промисов.	Колбэк может быть только один.

Таким образом, промисы позволяют улучшить порядок

кода и дают нам гибкость. Но это далеко не всё. Мы узнаем ещё много полезного в последующих главах.

Задачи

Можно ли "перевыполнить" промис?

Что выведет код ниже?

```
let promise = new Promise(function(resolve, reject) {
  resolve(1);

  setTimeout(() => resolve(2), 1000);
});

promise.then(alert);
```

К решению

Задержка на промисах

Встроенная функция `setTimeout` использует колбэк-функции. Создайте альтернативу, использующую промисы.

Функция `delay(ms)` должна возвращать промис, который перейдёт в состояние «выполнен» через `ms` миллисекунд, так чтобы мы могли добавить к нему `.then`:

```
function delay(ms) {
  // ваш код
}

delay(3000).then(() => alert('выполнилось через 3 секунды'));
```

К решению

Анимация круга с помощью промиса

Перепишите функцию `showCircle`, написанную в задании Анимация круга с помощью колбэка таким образом, чтобы она возвращала промис, вместо того чтобы принимать в аргументы функцию-callback.

Новое использование:

```
showCircle(150, 150, 100).then(div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

Возьмите решение из Анимация круга с помощью колбэка в качестве основы.

К решению

Цепочка промисов

Давайте вернёмся к ситуации из главы Введение: колбэки: у нас есть последовательность асинхронных задач, которые должны быть выполнены одна за другой. Например, речь может идти о загрузке скриптов. Как же грамотно реализовать это в коде?

Промисы предоставляют несколько способов решения подобной задачи.

В этой главе мы разберём цепочку промисов.

Она выглядит вот так:

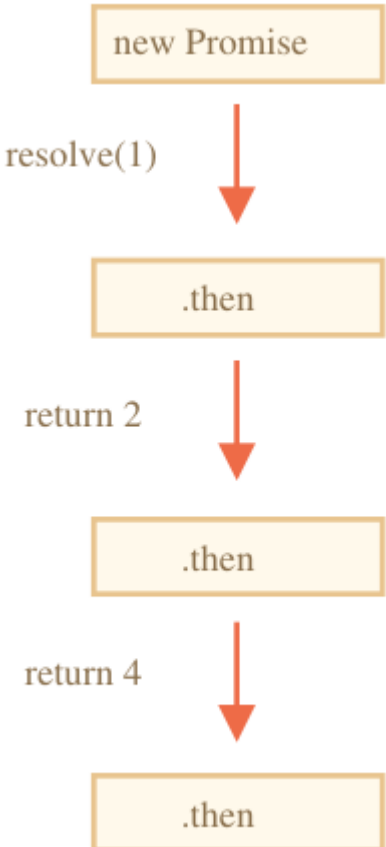
```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

Идея состоит в том, что результат первого промиса передаётся по цепочке обработчиков `.then`.

Поток выполнения такой:

1. Начальный промис успешно выполняется через 1 секунду `(*)`,
2. Затем вызывается обработчик в `.then (**)`.
3. Возвращаемое им значение передаётся дальше в следующий обработчик `.then (***)`
4. ...и так далее.

В итоге результат передаётся по цепочке обработчиков, и мы видим несколько `alert` подряд, которые выводят: `1 → 2 → 4`.



Всё это работает, потому что вызов `promise.then` тоже возвращает промис, так что мы можем вызвать на нём следующий `.then`.

Когда обработчик возвращает какое-то значение, то оно становится результатом выполнения соответствующего промиса и передаётся в следующий `.then`.

Классическая ошибка новичков: технически возможно добавить много обработчиков `.then` к единственному промису. Но это не цепочка.

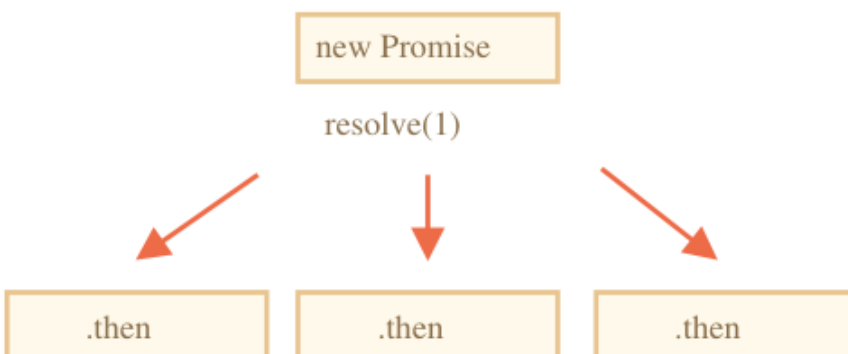
Например:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
```

```
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

Мы добавили несколько обработчиков к одному промису. Они не передают друг другу результаты своего выполнения, а действуют независимо.

Вот картина происходящего (сравните это с изображением цепочки промисов выше):



Все обработчики `.then` на одном и том же промисе получают одно и то же значение – результат выполнения того же самого промиса. Таким образом, в коде выше все `alert` показывают одно и то же: `1`.

На практике весьма редко требуется назначать несколько обработчиков одному промису. А вот цепочка промисов используется куда чаще.

Возвращаем промисы

Обработчик `handler`, переданный в `.then(handler)`, может вернуть промис.

В этом случае дальнейшие обработчики ожидают, пока он выполнится, и затем получают его результат.

Например:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1



}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

```
  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });
```

Здесь первый `.then` показывает `1` и возвращает новый промис `new Promise(…)` в строке `(*)`. Через одну секунду этот промис успешно выполняется, и его результат (аргумент в `resolve`, то есть `result * 2`) передаётся обработчику в следующем `.then`. Он

находится в строке `(**)`, показывает `2` и делает то же самое.

Таким образом, как и в предыдущем примере, выводятся 1 → 2 → 4, но сейчас между вызовами `alert` существует пауза в 1 секунду.

Возвращая промисы, мы можем строить цепочки из асинхронных действий.

Пример: loadScript

Давайте используем эту возможность вместе с промисифицированной функцией `loadScript`, созданной нами в предыдущей главе, чтобы загружать скрипты по очереди, последовательно:

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // вызовем функции, объявленные в загружаемых скриптах,
    // чтобы показать, что они действительно загрузились
    one();
    two();
    three();
  });
```

Этот же код можно переписать немного компактнее, используя стрелочные функции:

```
loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/
two.js"))
  .then(script => loadScript("/article/promise-chaining/
three.js"))
  .then(script => {
    // скрипты загружены, мы можем использовать объявленные в
```

```
них функции
    one();
    two();
    three();
  });
```

Здесь каждый вызов `loadScript` возвращает промис, и следующий обработчик в `.then` срабатывает, только когда этот промис завершается. Затем инициируется загрузка следующего скрипта и так далее. Таким образом, скрипты загружаются один за другим.

Мы можем добавить и другие асинхронные действия в цепочку. Обратите внимание, что наш код всё ещё «плоский», он «растёт» вниз, а не вправо. Нет никаких признаков «адской пирамиды вызовов».

Технически мы бы могли добавлять `.then` напрямую к каждому вызову `loadScript`, вот так:

```
loadScript("/article/promise-chaining/one.js").then(script1 =>
{
  loadScript("/article/promise-chaining/two.js").then(script2
=> {
    loadScript("/article/promise-chaining/
three.js").then(script3 => {
      // эта функция имеет доступ к переменным script1,
script2 и script3
      one();
      two();
      three();
    });
  });
});
```

Этот код делает то же самое: последовательно загружает 3 скрипта. Но он «растёт вправо», так что возникает такая же проблема, как и с колбэками.

Разработчики, которые не так давно начали использовать промисы, иногда не знают про цепочки и пишут код именно так, как показано выше. В целом, использование цепочек промисов предпочтительнее.

Иногда всё же приемлемо добавлять `.then` напрямую, чтобы вложенная в него функция имела доступ к внешней области видимости. В примере выше самая глубоко вложенная функция обратного вызова имеет доступ ко всем переменным `script1`, `script2`, `script3`. Но это скорее исключение, чем правило.

Thenable

Если быть более точными, обработчик может возвращать не именно промис, а любой объект, содержащий метод `.then`, такие объекты называют «thenable», и этот объект будет обработан как промис.

Смысл в том, что сторонние библиотеки могут создавать свои собственные совместимые с промисами объекты. Они могут иметь свои наборы методов и при этом быть совместимыми со встроенными промисами, так как реализуют метод `.then`.

Вот пример такого объекта:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // будет успешно выполнено с аргументом
this.num*2 через 1 секунду
    setTimeout(() => resolve(this.num * 2), 1000);
// (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {

  })
  .then(alert); // показывает 2 через 1000мс
```

```
    return new Thenable(result); // (*)
```

JavaScript проверяет объект, возвращаемый из обработчика `.then` в строке `(*)`: если у него имеется метод `then`, который можно вызвать, то этот метод вызывается, и в него передаются как аргументы встроенные функции `resolve` и `reject`, вызов одной из которых потом ожидается. В примере выше происходит вызов `resolve(2)` через 1 секунду `(**)`. Затем результат передаётся дальше по цепочке.

Это позволяет добавлять в цепочки промисов пользовательские объекты, не заставляя их наследовать от `Promise`.

Более сложный пример: fetch

Во фронтенд-разработке промисы часто используются, чтобы делать запросы по сети. Давайте рассмотрим один такой пример.

Мы будем использовать метод fetch, чтобы подгрузить информацию о пользователях с удалённого сервера. Этот метод имеет много опциональных параметров, разобранных в соответствующих разделах, но базовый синтаксис весьма прост:

```
let promise = fetch(url);
```

Этот код запрашивает по сети `url` и возвращает промис. Промис успешно выполняется и в свою очередь возвращает объект `response` после того, как удалённый сервер присылает заголовки ответа, но *до того, как весь ответ сервера полностью загружен*.

Чтобы прочитать полный ответ, надо вызвать метод

`response.text()`: он тоже возвращает промис, который выполняется, когда данные полностью загружены с удалённого сервера, и возвращает эти данные.

Код ниже запрашивает файл `user.json` и загружает его содержимое с сервера:

```
fetch('/article/promise-chaining/user.json')
  // .then в коде ниже выполняется, когда удалённый сервер
отвечает
  .then(function(response) {
    // response.text() возвращает новый промис,
    // который выполняется и возвращает полный ответ сервера,
    // когда он загрузится
    return response.text();
  })
  .then(function(text) {
    // ...и здесь содержимое полученного файла
    alert(text); // {"name": "iliakan", isAdmin: true}
  });
```

Есть также метод `response.json()`, который читает данные в формате JSON. Он больше подходит для нашего примера, так что давайте использовать его.

Мы также применим стрелочные функции для более компактной записи кода:

```
// то же самое, что и раньше, только теперь response.json()
читает данные в формате JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan, получили имя
пользователя
```

Теперь давайте что-нибудь сделаем с полученными данными о пользователе.

Например, мы можем послать запрос на GitHub, чтобы загрузить данные из профиля пользователя и показать его аватар:

```
// Запрашиваем user.json
fetch('/article/promise-chaining/user.json')
  // Загружаем данные в формате json
  .then(response => response.json())
  // Делаем запрос к GitHub
  .then(user => fetch(`https://api.github.com/users/
${user.name}`))
  // Загружаем ответ в формате json
  .then(response => response.json())
  // Показываем аватар (githubUser.avatar_url) в течение 3
секунд (возможно, с анимацией)
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

Код работает, детали реализации отражены в комментариях. Однако в нём есть одна потенциальная проблема, с которой часто сталкиваются новички.

Посмотрите на строку `(*)`: как мы можем предпринять какие-то действия *после* того, как аватар был показан и удалён? Например, мы бы хотели показывать форму редактирования пользователя или что-то ещё. Сейчас это невозможно.

Чтобы сделать наш код расширяемым, нам нужно возвращать ещё один промис, который выполняется после того, как завершается показ аватара.

Примерно так:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/
${user.name}`))
  .then(response => response.json())

    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);
```

```
    setTimeout(() => {
      img.remove();

    }, 3000);
  }))
  // срабатывает через 3 секунды
  .then(githubUser => alert(`Закончили показ
${githubUser.name}`));
   .then(githubUser => new
Promise(function(resolve, reject) { // (*)
      resolve(githubUser); // (**)
```

То есть, обработчик `.then` в строке `(*)` будет возвращать `new Promise`, который перейдёт в состояние «выполнен» только после того, как в `setTimeout` `(**)` будет вызвана `resolve(githubUser)`.

Соответственно, следующий по цепочке `.then` будет ждать этого.

Как правило, все асинхронные действия должны возвращать промис.

Это позволяет планировать после него какие-то дополнительные действия. Даже если эта возможность не нужна прямо сейчас, она может понадобиться в будущем.

И, наконец, давайте разобьём написанный код на отдельные функции, пригодные для повторного использования:

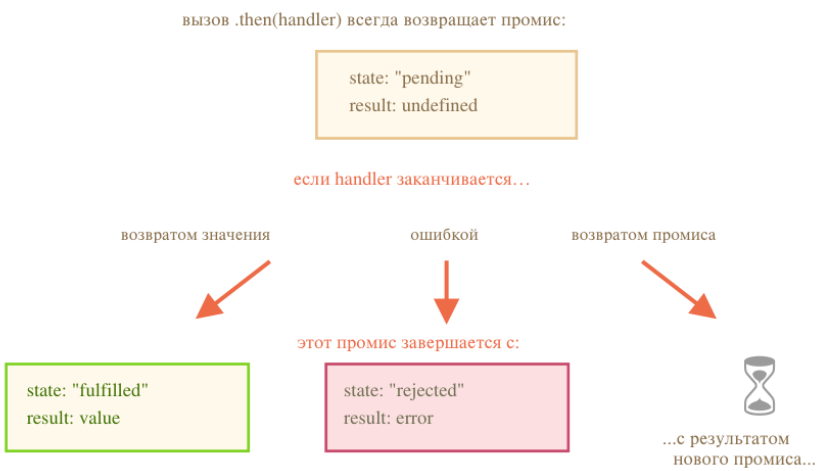
```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return fetch(`https://api.github.com/users/${name}`)
    .then(response => response.json());
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
```

```
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Используем их:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Показ аватара ${githubUser.name} завершён`));
  // ...
```

Итого

Если обработчик в `.then` (или в `catch/finally`, без разницы) возвращает промис, последующие элементы цепочки ждут, пока этот промис выполнится. Когда это происходит, результат его выполнения (или ошибка) передаётся дальше.

Вот полная картина происходящего:



Задачи

Промисы: сравните then и catch

Являются ли фрагменты кода ниже эквивалентными? Другими словами, ведут ли они себя одинаково во всех обстоятельствах, для всех переданных им обработчиков?

```
promise.then(f1).catch(f2);
```

Против:

```
promise.then(f1, f2);
```

К решению

Промисы: обработка ошибок

Цепочки промисов отлично подходят для перехвата ошибок. Если промис завершается с ошибкой, то управление переходит в ближайший обработчик ошибок. На практике это очень удобно.

Например, в представленном ниже примере для `fetch` указана неправильная ссылка (сайт не существует), и `.catch` перехватывает ошибку:

```
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch
(текст может отличаться)
fetch('https://no-such-server.blabla') //
ошибка
```

Как видно, `.catch` не обязательно должен быть сразу после ошибки, он может быть далее, после одного или даже нескольких `.then`

Или, может быть, с сервером всё в порядке, но в ответе мы получим некорректный JSON. Самый лёгкий путь перехватить все ошибки – это добавить `.catch` в конец цепочки:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/
${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise((resolve, reject) => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Если все в порядке, то такой `.catch` вообще не выполнится. Но если любой из промисов будет отклонён (проблемы с сетью или некорректная json-строка, или что угодно другое), то ошибка будет перехвачена.

Неявный try…catch

Вокруг функции промиса и обработчиков находится "невидимый `try..catch`". Если происходит исключение, то оно перехватывается, и промис считается отклонённым с этой ошибкой.

Например, этот код:

```
new Promise((resolve, reject) => {

}).catch(alert); // Error: Ошибка!
  throw new Error("Ошибка!");
```

…Работает так же, как и этот:

```
new Promise((resolve, reject) => {

}).catch(alert); // Error: Ошибка!
  reject(new Error("Ошибка!"));
```

"Невидимый `try..catch`" вокруг промиса автоматически перехватывает ошибку и превращает её в отклонённый промис.

Это работает не только в функции промиса, но и в обработчиках. Если мы бросим ошибку (`throw`) из обработчика (`.then`), то промис будет считаться отклонённым, и управление перейдёт к ближайшему обработчику ошибок.

Пример:

```
new Promise((resolve, reject) => {
  resolve("ок");
}).then((result) => {

}).catch(alert); // Error: Ошибка!
  throw new Error("Ошибка!"); // генерируем
ошибку
```

Это происходит для всех ошибок, не только для тех, которые вызваны оператором `throw`. Например, программная ошибка:

```
new Promise((resolve, reject) => {
  resolve("ок");
}).then((result) => {

}).catch(alert); // ReferenceError: blabla is not defined
```

```
  blabla(); // нет такой функции
```

Финальный `.catch` перехватывает как промисы, в которых вызван `reject`, так и случайные ошибки в обработчиках.

Пробрасывание ошибок

Как мы уже заметили, `.catch` ведёт себя как `try..catch`. Мы можем иметь столько обработчиков `.then`, сколько мы хотим, и затем использовать один `.catch` в конце, чтобы перехватить ошибки из всех обработчиков.

В обычном `try..catch` мы можем проанализировать ошибку и повторно пробросить дальше, если не можем её обработать. То же самое возможно для промисов.

Если мы пробросим (`throw`) ошибку внутри блока `.catch`, то управление перейдёт к следующему ближайшему обработчику ошибок. А если мы обработаем ошибку и завершим работу обработчика нормально, то продолжит работу ближайший успешный обработчик `.then`.

В примере ниже `.catch` успешно обрабатывает ошибку:

```
// the execution: catch -> then
new Promise((resolve, reject) => {

  throw new Error("Ошибка!");

}).catch(function(error) {

  alert("Ошибка обработана, продолжить работу");

}).then(() => alert("Управление перейдёт в следующий then"));
```

Здесь блок `.catch` завершается нормально. Поэтому вызывается следующий успешный обработчик `.then`.

В примере ниже мы видим другую ситуацию с блоком `.catch`. Обработчик `(*)` перехватывает ошибку и не может обработать её (например, он знает как обработать только `URIError`), поэтому ошибка пробрасывается далее:

```
// the execution: catch -> catch -> then
new Promise((resolve, reject) => {

  throw new Error("Ошибка!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // обрабатываем ошибку
  } else {
    alert("Не могу обработать ошибку");


  }

}).then(function() {
  /* не выполнится */
}).catch(error => { // (**)

  alert(`Неизвестная ошибка: ${error}`);
  // ничего не возвращаем => выполнение продолжается в
нормальном режиме

});
    throw error; // пробрасывает эту или
другую ошибку в следующий catch
```

Управление переходит от первого блока `.catch (*)` к следующему `(**)`, вниз по цепочке.

Необработанные ошибки

Что произойдёт, если ошибка не будет обработана? Например, мы просто забыли добавить `.catch` в конец цепочки, как здесь:

```
new Promise(function() {
  noSuchFunction(); // Ошибка (нет такой функции)
})
  .then(() => {
    // обработчики .then, один или более
  }); // без .catch в самом конце!
```

В случае ошибки выполнение должно перейти к ближайшему обработчику ошибок. Но в примере выше нет никакого обработчика. Поэтому ошибка как бы «застревает», её некому обработать.

На практике, как и при обычных необработанных ошибках в коде, это означает, что что-то пошло сильно не так.

Что происходит, когда обычная ошибка не перехвачена `try..catch`? Скрипт умирает с сообщением в консоли. Похожее происходит и в случае необработанной ошибки промиса.

JavaScript-движок отслеживает такие ситуации и генерирует в этом случае глобальную ошибку. Вы можете увидеть её в консоли, если запустите пример выше.

В браузере мы можем поймать такие ошибки, используя событие `unhandledrejection`:

```
new Promise(function() {
  throw new Error("Ошибка!");
}); // нет обработчика ошибок
window.addEventListener('unhandledrejection',
```

```
function(event) {
  // объект события имеет два специальных
свойства:
  alert(event.promise); // [object Promise] -
промис, который сгенерировал ошибку
  alert(event.reason); // Error: Ошибка! -
объект ошибки, которая не была обработана
});
```

Это событие является частью стандарта HTML.

Если происходит ошибка, и отсутствует её обработчик, то генерируется событие `unhandledrejection`, и соответствующий объект `event` содержит информацию об ошибке.

Обычно такие ошибки неустранимы, поэтому лучше всего – информировать пользователя о проблеме и, возможно, отправить информацию об ошибке на сервер.

В не-браузерных средах, таких как Node.js, есть другие способы отслеживания необработанных ошибок.

Итого

`.catch` перехватывает все виды ошибок в промисах: будь то вызов `reject()` или ошибка, брошенная в обработчике при помощи `throw`.

Необходимо размещать `.catch` там, где мы хотим обработать ошибки и знаем, как это сделать. Обработчик может проанализировать ошибку (могут быть полезны пользовательские классы ошибок) и пробросить её, если ничего не знает о ней (возможно, это программная ошибка).

Можно и совсем не использовать `.catch`, если нет нормального способа восстановиться после ошибки.

В любом случае нам следует использовать обработчик

события `unhandledrejection` (для браузеров и аналог для других окружений), чтобы отслеживать необработанные ошибки и информировать о них пользователя (и, возможно, наш сервер), благодаря чему наше приложение никогда не будет «просто умирать».

Задачи

Ошибка в setTimeout

Что вы думаете? Выполнится ли `.catch`? Поясните свой ответ.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

К решению

Promise API

В классе `Promise` есть 5 статических методов. Давайте познакомимся с ними.

Promise.all

Допустим, нам нужно запустить множество промисов параллельно и дождаться, пока все они выполнятся.

Например, параллельно загрузить несколько файлов и обработать результат, когда он готов.

Для этого как раз и пригодится `Promise.all`.

Синтаксис:

```
let promise = Promise.all([...промисы...]);
```

Метод `Promise.all` принимает массив промисов (может принимать любой перебираемый объект, но обычно используется массив) и возвращает новый промис.

Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов.

Например, `Promise.all`, представленный ниже, выполнится спустя 3 секунды, его результатом будет массив `[1, 2, 3]`:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)),
// 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)),
// 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))
// 3
]).then(alert); // когда все промисы выполнятся, результат
будет 1,2,3
// каждый промис даёт элемент массива
```

Обратите внимание, что порядок элементов массива в точности соответствует порядку исходных промисов. Даже если первый промис будет выполняться дольше всех, его результат всё равно будет первым в массиве.

Часто применяемый трюк – пропустить массив данных через map-функцию, которая для каждого элемента создаст задачу-промис, и затем обернёт получившийся массив в `Promise.all`.

Например, если у нас есть массив ссылок, то мы можем загрузить их вот так:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// Преобразуем каждый URL в промис, возвращённый fetch
let requests = urls.map(url => fetch(url));

// Promise.all будет ожидать выполнения всех промисов
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

А вот пример побольше, с получением информации о пользователях GitHub по их логинам из массива (мы могли бы получать массив товаров по их идентификаторам, логика та же):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://
api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // все промисы успешно завершены
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // покажет
200 для каждой ссылки
    }

    return responses;
  })
  // преобразовать массив ответов response в response.json(),
  // чтобы прочитать содержимое каждого
  .then(responses => Promise.all(responses.map(r =>
r.json())))
  // все JSON-ответы обработаны, users - массив с результатами
  .then(users => users.forEach(user => alert(user.name)));
```

Если любой из промисов завершится с ошибкой, то

промис, возвращённый `Promise.all`, немедленно завершается с этой ошибкой.

Например:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() =>
resolve(1), 1000)),

  new Promise((resolve, reject) => setTimeout(() =>
resolve(3), 3000))
]).catch(alert); // Error: Ошибка!
  new Promise((resolve, reject) =>
setTimeout(() => reject(new Error("Ошибка!")),
2000)),
```

Здесь второй промис завершится с ошибкой через 2 секунды. Это приведёт к немедленной ошибке в `Promise.all`, так что выполнится `.catch`: ошибка этого промиса становится ошибкой всего `Promise.all`.

В случае ошибки, остальные результаты игнорируются

Если один промис завершается с ошибкой, то весь `Promise.all` завершается с ней, полностью забывая про остальные промисы в списке. Их результаты игнорируются.

Например, если сделано несколько вызовов `fetch`, как в примере выше, и один не прошёл, то остальные будут всё ещё выполняться, но `Promise.all` за ними уже не смотрит. Скорее всего, они так или иначе завершатся, но их результаты будут проигнорированы.

`Promise.all` ничего не делает для их отмены, так как в промисах вообще нет концепции «отмены». В главе Fetch: прерывание запроса мы

рассмотрим `AbortController`, который помогает с этим, но он не является частью Promise API.

`Promise.all(iterable)` разрешает передавать не-промисы в `iterable` (перебираемом объекте)

Обычно, `Promise.all(...)` принимает перебираемый объект промисов (чаще всего массив). Но если любой из этих объектов не является промисом, он передаётся в итоговый массив «как есть».

Например, здесь результат: `[1, 2, 3]`

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2,
  3
]).then(alert); // 1, 2, 3
```

Таким образом, мы можем передавать уже готовые значения, которые не являются промисами, в `Promise.all`, иногда это бывает удобно.

Promise.allSettled

Новая возможность

Эта возможность была добавлена в язык

недавно. В старых браузерах может понадобиться полифил.

`Promise.all` завершается с ошибкой, если она возникает в любом из переданных промисов. Это подходит для ситуаций «всё или ничего», когда нам нужны *все* результаты для продолжения:

```
Promise.all([
  fetch('/template.html'),
  fetch('/style.css'),
  fetch('/data.json')
]).then(render); // методу render нужны результаты всех fetch
```

Метод `Promise.allSettled` всегда ждёт завершения всех промисов. В массиве результатов будет

- `{status:"fulfilled", value:результат}` для успешных завершений,
- `{status:"rejected", reason:ошибка}` для ошибок.

Например, мы хотели бы загрузить информацию о множестве пользователей. Даже если в каком-то запросе ошибка, нас всё равно интересуют остальные.

Используем для этого `Promise.allSettled`:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://no-such-url'
];

Promise.allSettled(urls.map(url => fetch(url)))
  .then(results => { // (*)
    results.forEach((result, num) => {
      if (result.status == "fulfilled") {
        alert(`${urls[num]}: ${result.value.status}`);
      }
      if (result.status == "rejected") {
```

```
        alert(`${urls[num]}: ${result.reason}`);
      }
    });
  });
```

Массив `results` в строке `(*)` будет таким:

```
[
  {status: 'fulfilled', value: ...объект ответа...},
  {status: 'fulfilled', value: ...объект ответа...},
  {status: 'rejected', reason: ...объект ошибки...}
]
```

То есть, для каждого промиса у нас есть его статус и значение/ошибка.

Полифил

Если браузер не поддерживает `Promise.allSettled`, для него легко сделать полифил:

```
if(!Promise.allSettled) {
  Promise.allSettled = function(promises) {
    return Promise.all(promises.map(p =>
Promise.resolve(p).then(value => ({
      status: 'fulfilled',
      value: value
    }), error => ({
      status: 'rejected',
      reason: error
    }))));
  };
}
```

В этом коде `promises.map` берёт аргументы, превращает их в промисы (на всякий случай) и добавляет каждому обработчик `.then`.

Этот обработчик превращает успешный результат `value` в `{state:'fulfilled', value: value}`, а ошибку `error` в `{state:'rejected', reason: error}`. Это как раз и есть формат результатов `Promise.allSettled`.

Затем мы можем использовать `Promise.allSettled`, чтобы получить результаты *всех* промисов, даже если при выполнении какого-то возникнет ошибка.

Promise.race

Метод очень похож на `Promise.all`, но ждёт только первый промис, из которого берёт результат (или ошибку).

Синтаксис:

```
let promise = Promise.race(iterable);
```

Например, тут результат будет `1`:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() =>
resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new
Error("Ошибка!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() =>
resolve(3), 3000))
]).then(alert); // 1
```

Быстрее всех выполнился первый промис, он и дал результат. После этого остальные промисы игнорируются.

Promise.resolve/reject

Методы `Promise.resolve` и `Promise.reject` редко используются в современном коде, так как синтаксис `async/await` (мы рассмотрим его чуть позже) делает его, в общем-то, не нужным.

Мы рассмотрим их здесь для полноты картины, а также для тех, кто по каким-то причинам не может использовать `async/await`.

`Promise.resolve(value)` создаёт успешно выполненный промис с результатом `value`.

То же самое, что:

```
let promise = new Promise(resolve => resolve(value));
```

Этот метод используют для совместимости: когда ожидается, что функция возвратит именно промис.

Например, функция `loadCached` ниже загружает URL и запоминает (кеширует) его содержимое. При будущих вызовах с тем же URL он тут же читает предыдущее содержимое из кеша, но использует `Promise.resolve`, чтобы сделать из него промис, для того, чтобы возвращаемое значение всегда было промисом:

```
let cache = new Map();

function loadCached(url) {
  if (cache.has(url)) {

  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
     return Promise.resolve(cache.get(url)); //
(*)
```

Мы можем писать `loadCached(url).then(…)`, потому что функция `loadCached` всегда возвращает промис. Мы всегда можем использовать `.then` после `loadCached`. Это и есть цель использования `Promise.resolve` в строке `(*)`.

Promise.reject

`Promise.reject(error)` создаёт промис, завершённый с ошибкой `error`.

То же самое, что:

```
let promise = new Promise((resolve, reject) => reject(error));
```

На практике этот метод почти никогда не используется.

Итого

Мы ознакомились с пятью статическими методами класса `Promise`:

1. `Promise.all(promises)` – ожидает выполнения всех промисов и возвращает массив с результатами. Если любой из указанных промисов вернёт ошибку, то результатом работы `Promise.all` будет эта ошибка, результаты остальных промисов будут игнорироваться.
2. `Promise.allSettled(promises)` (добавлен недавно) – ждёт, пока все промисы завершатся и возвращает их результаты в виде массива с объектами, у каждого объекта два свойства:

   `state`: `"fulfilled"`, если выполнен успешно или `"rejected"`, если ошибка,

   `value` – результат, если успешно или `reason` – ошибка, если нет.
5. `Promise.race(promises)` – ожидает первый выполненный промис, который становится его результатом, остальные игнорируются.
6. `Promise.resolve(value)` – возвращает успешно выполнившийся промис с результатом `value`.
7. `Promise.reject(error)` – возвращает промис с ошибкой `error`.

Из всех перечисленных методов, самый часто используемый – это, пожалуй, `Promise.all`.

Промисификация

Промисификация – это длинное слово для простого преобразования. Мы берём функцию, которая принимает колбэк и меняем её, чтобы она вместо этого возвращала промис.

Такие преобразования часто необходимы в реальной жизни, так как многие функции и библиотеки основаны на колбэках, а использование промисов более удобно, поэтому есть смысл «промисифицировать» их.

Например, у нас есть `loadScript(src, callback)` из главы Введение: колбэки.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Ошибка загрузки
скрипта ${src}`));

  document.head.append(script);
}

// использование:
// loadScript('path/script.js', (err, script) => {...})
```

Давайте промисифицируем её. Новая функция `loadScriptPromise(src)` будет делать то же самое, но будет принимать только `src` (не `callback`) и возвращать промис.

```
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
```

```
      else resolve(script);
    });
  })
}

// использование:
// loadScriptPromise('path/script.js').then(...)
```

Теперь `loadScriptPromise` хорошо вписывается в код, основанный на промисах.

Как видно, она передаёт всю работу исходной функции `loadScript`, предоставляя ей колбэк, по вызову которого происходит `resolve/reject` промиса.

На практике нам, скорее всего, понадобится промисифицировать не одну функцию, поэтому имеет смысл сделать для этого специальную «функцию-помощник».

Мы назовём её `promisify(f)` – она принимает функцию для промисификации `f` и возвращает функцию-обёртку.

Эта функция-обёртка делает то же самое, что и код выше: возвращает промис и передаёт вызов оригинальной `f`, отслеживая результат в своём колбэке:

```
function promisify(f) {
  return function (...args) { // возвращает функцию-обёртку
    return new Promise((resolve, reject) => {
      function callback(err, result) { // наш специальный
колбэк для f
        if (err) {
          return reject(err);
        } else {
          resolve(result);
        }
      }

      args.push(callback); // добавляем колбэк в конец
аргументов f

      f.call(this, ...args); // вызываем оригинальную функцию
    });
  };
};
```

```
// использование:
let loadScriptPromise = promisify(loadScript);
loadScriptPromise(...).then(...);
```

Здесь мы предполагаем, что исходная функция ожидает колбэк с двумя аргументами `(err, result)`. Это то, с чем мы чаще всего сталкиваемся. Тогда наш колбэк – в правильном формате, и `promisify` отлично работает для такого случая.

Но что, если исходная `f` ожидает колбэк с большим количеством аргументов `callback(err, res1, res2, ...)`?

Ниже описана улучшенная функция `promisify`: при вызове `promisify(f, true)` результатом промиса будет массив результатов `[res1, res2, ...]`:

```
// promisify(f, true), чтобы получить массив результатов
function promisify(f, manyArgs = false) {
  return function (...args) {
    return new Promise((resolve, reject) => {
      function ) { // наш специальный колбэк для f
        if (err) {
          return reject(err);
        } else {
          // делаем resolve для всех results колбэка, если
задано manyArgs

        }
      }

      args.push(callback);

      f.call(this, ...args);
    });
  };
};

// использование:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
callback
(err, ...results
resolve
```

```
(manyArgs ? results : results[0]);
```

Для более экзотических форматов колбэка, например без `err: callback(result)`, мы можем промисифицировать функции без помощника, «вручную».

Также существуют модули с более гибкой промисификацией, например, es6-promisify или встроенная функция `util.promisify` в Node.js.

> **На заметку:**
>
> Промисификация – это отличный подход, особенно, если вы будете использовать `async/await` (см. следующую главу), но она не является тотальной заменой любых колбэков.
>
> Помните, промис может иметь только один результат, но колбэк технически может вызываться сколько угодно раз.
>
> Поэтому промисификация используется для функций, которые вызывают колбэк только один раз. Последующие вызовы колбэка будут проигнорированы.

Микрозадачи

Обработчики промисов `.then/.catch/.finally` всегда асинхронны.

Даже когда промис сразу же выполнен, код в строках *ниже* `.then/.catch/.finally` будет запущен до этих обработчиков.

Вот демо:

```
let promise = Promise.resolve();

promise.then(() => alert("промис выполнен"));

alert("код выполнен"); // этот alert показывается первым
```

Если вы запустите его, сначала вы увидите `код выполнен`, а потом `промис выполнен`.

Это странно, потому что промис определённо был выполнен с самого начала.

Почему `.then` срабатывает позже? Что происходит?

Очередь микрозадач

Асинхронные задачи требуют правильного управления. Для этого стандарт предусматривает внутреннюю очередь `PromiseJobs`, более известную как «очередь микрозадач (microtask queue)» (термин V8).

Как сказано в спецификации:

> Очередь определяется как первым-пришёл-первым-ушёл (FIFO): задачи, попавшие в очередь первыми, выполняются тоже первыми.
>
> Выполнение задачи происходит только в том случае, если ничего больше не запущено.

Или, проще говоря, когда промис выполнен, его обработчики `.then/catch/finally` попадают в очередь. Они пока не выполняются. Движок JavaScript берёт задачу из очереди и выполняет её, когда он освободится от выполнения текущего кода.

Вот почему сообщение «код выполнен» в примере выше будет показано первым.

Обработчики промисов всегда проходят через эту внутреннюю очередь.

Если есть цепочка с несколькими `.then/catch/finally`, то каждый из них выполняется асинхронно. То есть сначала ставится в очередь, а потом выполняется, когда выполнение текущего кода завершено и добавленные ранее в очередь обработчики выполнены.

Но что если порядок имеет значение для нас? Как мы можем вывести `код выполнен` после `промис выполнен`?

Легко, используя `.then`:

```
Promise.resolve()
  .then(() => alert("промис выполнен!"))
  .then(() => alert("код выполнен"));
```

Теперь порядок стал таким, как было задумано.

Необработанные ошибки

Помните «необработанные ошибки» из главы Промисы: обработка ошибок?

Теперь мы можем описать, как именно JavaScript

понимает, что ошибка не обработана.

"Необработанная ошибка" возникает в случае, если ошибка промиса не обрабатывается в конце очереди микрозадач.

Обычно, если мы ожидаем ошибку, мы добавляем `.catch` в конец цепочки промисов, чтобы обработать её:

```
let promise = Promise.reject(new Error("Ошибка в промисе!"));


// не запустится, ошибка обработана
window.addEventListener('unhandledrejection', event => {
  alert(event.reason);
});
promise.catch(err => alert('поймана!'));
```

…Но если мы забудем добавить `.catch`, то, когда очередь микрозадач опустеет, движок сгенерирует событие:

```
let promise = Promise.reject(new Error("Ошибка в промисе!"));

// Ошибка в промисе!
window.addEventListener('unhandledrejection', event =>
alert(event.reason));
```

А что, если мы поймаем ошибку, но позже? Вот так:

```
let promise = Promise.reject(new Error("Ошибка в промисе!"));



// Ошибка в промисе!
window.addEventListener('unhandledrejection', event =>
alert(event.reason));
setTimeout(() => promise.catch(err =>
alert('поймана')), 1000);
```

Теперь, при запуске, мы сначала увидим «Ошибка в

промисе!», а затем «поймана».

Если бы мы не знали про очередь микрозадач, то могли бы удивиться: «Почему сработал обработчик `unhandledrejection`? Мы же поймали ошибку!».

Но теперь мы понимаем, что событие `unhandledrejection` возникает, когда очередь микрозадач завершена: движок проверяет все промисы и, если какой-либо из них в состоянии «rejected», то генерируется это событие.

В примере выше `.catch`, добавленный в `setTimeout`, также срабатывает, но позже, уже после возникновения `unhandledrejection`, так что это ни на что не влияет.

Итого

Обработка промисов всегда асинхронная, т.к. все действия промисов проходят через внутреннюю очередь «promise jobs», так называемую «очередь микрозадач (microtask queue)» (термин v8).

Таким образом, обработчики `.then/catch/finally` вызываются после выполнения текущего кода.

Если нам нужно гарантировать выполнение какого-то кода после `.then/catch/finally`, то лучше всего добавить его вызов в цепочку `.then`.

В большинстве движков JavaScript, включая браузеры и Node.js, микрозадачи тесно связаны с так называемым «событийным циклом» и «макрозадачами». Так как они не связаны напрямую с промисами, то рассматриваются в другой части учебника, в главе Событийный цикл: микрозадачи и макрозадачи.

Async/await

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он удивительно прост для понимания и использования.

Асинхронные функции

Начнём с ключевого слова `async`. Оно ставится перед функцией, вот так:

```
async function f() {
  return 1;
}
```

У слова `async` один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

Например, эта функция возвратит выполненный промис с результатом `1`:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

Можно и явно вернуть промис, результат будет одинаковым:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Так что ключевое слово `async` перед функцией гарантирует, что эта функция в любом случае вернёт

промис. Согласитесь, достаточно просто? Но это ещё не всё. Есть другое ключевое слово – `await`, которое можно использовать только внутри `async`-функций.

Await

Синтаксис:

```
// работает только внутри async-функций
let value = await promise;
```

Ключевое слово `await` заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от `await` не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

В этом примере промис успешно выполнится через 1 секунду:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
  });



  alert(result); // "готово!"
}

f();
   let result = await promise; // будет ждать,
пока промис не выполнится (*)
```

В данном примере выполнение функции остановится на строке `(*)` до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную `result` будет записан результат выполнения промиса, и браузер отобразит alert-окно

«готово!».

Обратите внимание, хотя `await` и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.

По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем `promise.then`.

`await` нельзя использовать в обычных функциях

Если мы попробуем использовать `await` внутри функции, объявленной без `async`, получим синтаксическую ошибку:

```
function f() {
  let promise = Promise.resolve(1);

}
  let result = await promise; // SyntaxError
```

Ошибки не будет, если мы укажем ключевое слово `async` перед объявлением функции. Как было сказано раньше, `await` можно использовать только внутри `async`–функций.

Давайте перепишем пример `showAvatar()` из раздела Цепочка промисов с помощью `async/await`:

1. Нам нужно заменить вызовы `.then` на `await`.
2. И добавить ключевое слово `async` перед объявлением функции.

```
async function showAvatar() {

  // запрашиваем JSON с данными пользователя
  let response = await fetch('/article/promise-chaining/
user.json');
  let user = await response.json();

  // запрашиваем информацию об этом пользователе из github
  let githubResponse = await fetch(`https://api.github.com/
users/${user.name}`);
  let githubUser = await githubResponse.json();

  // отображаем аватар пользователя
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // ждём 3 секунды и затем скрываем аватар
  await new Promise((resolve, reject) => setTimeout(resolve,
3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Получилось очень просто и читаемо, правда? Гораздо лучше, чем раньше.

`await` нельзя использовать на верхнем уровне вложенности

Программисты, узнав об `await`, часто пытаются использовать эту возможность на верхнем уровне вложенности (вне тела функции). Но из-за того, что `await` работает только внутри `async`–функций, так сделать не получится:

```
// SyntaxError на верхнем уровне вложенности
let response = await fetch('/article/promise-
chaining/user.json');
```

```
let user = await response.json();
```

Можно обернуть этот код в анонимную `async`–функцию, тогда всё заработает:

```
(async () => {
  let response = await fetch('/article/promise-
chaining/user.json');
  let user = await response.json();
  ...
})();
```

`await` работает с «thenable»–объектами

Как и `promise.then`, `await` позволяет работать с промис–совместимыми объектами. Идея в том, что если у объекта можно вызвать метод `then`, этого достаточно, чтобы использовать его с `await`.

В примере ниже, экземпляры класса `Thenable` будут работать вместе с `await`:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve);
    // выполнить resolve со значением this.num * 2
через 1000мс
    setTimeout(() => resolve(this.num * 2), 1000);
// (*)
  }
};

async function f() {
  // код будет ждать 1 секунду,
  // после чего значение result станет равным 2
  let result = await new Thenable(1);
  alert(result);
}
```

```
f();
```

Когда `await` получает объект с `.then`, не являющийся промисом, JavaScript автоматически запускает этот метод, передавая ему аргументы – встроенные функции `resolve` и `reject`. Затем `await` приостановит дальнейшее выполнение кода, пока любая из этих функций не будет вызвана (в примере это строка `(*)`). После чего выполнение кода продолжится с результатом `resolve` или `reject` соответственно.

Асинхронные методы классов

Для объявления асинхронного метода достаточно написать `async` перед именем:

```
class Waiter {

    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1
  async wait() {
```

Как и в случае с асинхронными функциями, такой метод гарантированно возвращает промис, и в его теле можно использовать `await`.

Обработка ошибок

Когда промис завершается успешно, `await promise`

возвращает результат. Когда завершается с ошибкой – будет выброшено исключение. Как если бы на этом месте находилось выражение `throw`.

Такой код:

```
async function f() {

}
  await Promise.reject(new Error("Упс!"));
```

Делает то же самое, что и такой:

```
async function f() {

}
  throw new Error("Упс!");
```

Но есть отличие: на практике промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем `await` выбросит исключение.

Такие ошибки можно ловить, используя `try..catch`, как с обычным `throw`:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {

  }
}

f();
     alert(err); // TypeError: failed to fetch
```

В случае ошибки выполнение `try` прерывается и управление прыгает в начало блока `catch`. Блоком `try` можно обернуть несколько строк:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // перехватит любую ошибку в блоке try: и в fetch, и в
response.json
    alert(err);
  }
}

f();
```

Если у нас нет `try..catch`, асинхронная функция будет возвращать завершившийся с ошибкой промис (в состоянии `rejected`). В этом случае мы можем использовать метод `.catch` промиса, чтобы обработать ошибку:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() вернёт промис в состоянии rejected
f().catch(alert); // TypeError: failed to
fetch // (*)
```

Если забыть добавить `.catch`, то будет сгенерирована ошибка «Uncaught promise error» и информация об этом будет выведена в консоль. Такие ошибки можно поймать глобальным обработчиком, о чём подробно написано в разделе Промисы: обработка ошибок.

async/await и promise.then/catch

При работе с `async/await`, `.then` используется нечасто, так как `await` автоматически ожидает завершения выполнения промиса. В этом случае обычно (но не всегда)

гораздо удобнее перехватывать ошибки, используя `try..catch`, нежели чем `.catch`.

Но на верхнем уровне вложенности (вне `async`–функций) `await` использовать нельзя, поэтому `.then/catch` для обработки финального результата или ошибок – обычная практика.

Так сделано в строке `(*)` в примере выше.

`async/await` отлично работает с `Promise.all`

Когда необходимо подождать несколько промисов одновременно, можно обернуть их в `Promise.all`, и затем `await`:

```
// await будет ждать массив с результатами выполнения всех промисов
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

В случае ошибки она будет передаваться как обычно: от завершившегося с ошибкой промиса к `Promise.all`. А после будет сгенерировано исключение, которое можно отловить, обернув выражение в `try..catch`.

Итого

Ключевое слово `async` перед объявлением функции:

1. Обязывает её всегда возвращать промис.

2. Позволяет использовать `await` в теле этой функции.

Ключевое слово `await` перед промисом заставит JavaScript дождаться его выполнения, после чего:

1. Если промис завершается с ошибкой, будет сгенерировано исключение, как если бы на этом месте находилось `throw`.
2. Иначе вернётся результат промиса.

Вместе они предоставляют отличный каркас для написания асинхронного кода. Такой код легко и писать, и читать.

Хотя при работе с `async/await` можно обходиться без `promise.then/catch`, иногда всё-таки приходится использовать эти методы (на верхнем уровне вложенности, например). Также `await` отлично работает в сочетании с `Promise.all`, если необходимо выполнить несколько задач параллельно.

Задачи

Перепишите, используя async/await

Перепишите один из примеров раздела Цепочка промисов, используя `async/await` вместо `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
```

```
    })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // Error: 404
```

К решению

Перепишите, используя async/await

Ниже пример из раздела Цепочка промисов, перепишите его, используя `async/await` вместо `.then/catch`.

В функции `demoGithubUser` замените рекурсию на цикл: используя `async/await`, сделать это будет просто.

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
}

// Запрашивать логин, пока github не вернёт существующего пользователя.
function demoGithubUser() {
  let name = prompt("Введите логин?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Полное имя: ${user.name}.`);
      return user;
```

```
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status ==
404) {
        alert("Такого пользователя не существует, пожалуйста,
повторите ввод.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

К решению

Вызовите async–функцию из "обычной"

Есть «обычная» функция. Как можно внутри неё получить результат выполнения `async`–функции?

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve, 1000));

  return 10;
}

function f() {
  // ...что здесь написать?
  // чтобы вызвать wait() и дождаться результата "10" от
async-функции
  // не забывайте, здесь нельзя использовать "await"
}
```

P.S. Технически задача очень простая, но этот вопрос часто задают разработчики, недавно познакомившиеся с async/await.

К решению

Генераторы, продвинутая итерация

Генераторы

Обычные функции возвращают только одно-единственное значение (или ничего).

Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

Функция-генератор

Для объявления генератора используется специальная синтаксическая конструкция: `function*`, которая называется «функция-генератор».

Выглядит она так:

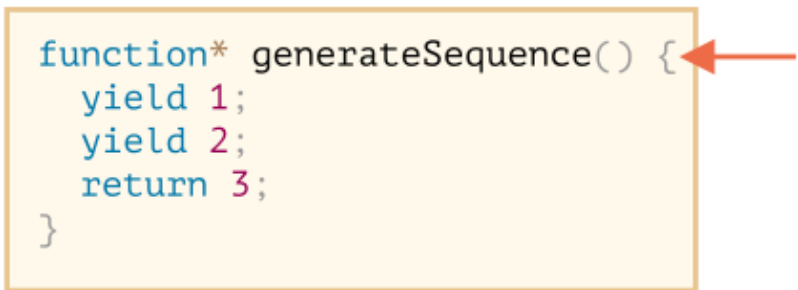
```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор», для управления её выполнением.

Вот, посмотрите:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

// "функция-генератор" создаёт объект "генератор"
let generator = generateSequence();
alert(generator); // [object Generator]
```

Выполнение кода функции ещё не началось:



Основным методом генератора является `next()`. При вызове он запускает выполнение кода до ближайшей инструкции `yield <значение>` (значение может отсутствовать, в этом случае оно предполагается равным `undefined`). По достижении `yield` выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода `next()` всегда является объект с двумя свойствами:

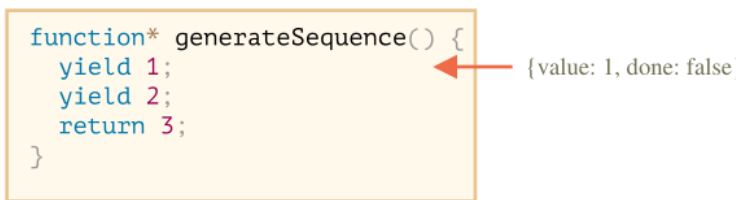
- `value`: значение из `yield`.
- `done`: `true`, если выполнение функции завершено, иначе `false`.

Например, здесь мы создаём генератор и получаем первое из возвращаемых им значений:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();
```

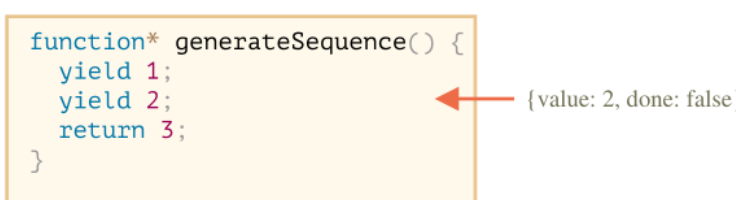
```
alert(JSON.stringify(one)); // {value: 1, done: false}
let one = generator.next();
```

На данный момент мы получили только первое значение, выполнение функции остановлено на второй строке:



Повторный вызов `generator.next()` возобновит выполнение кода и вернёт результат следующего `yield`:

```
let two = generator.next();

alert(JSON.stringify(two)); // {value: 2, done: false}
```



И, наконец, последний вызов завершит выполнение функции и вернёт результат `return`:

```
let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, }
done: true
```

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}
```


{value: 3, done: true}

Сейчас генератор полностью выполнен. Мы можем увидеть это по свойству `done:true` и обработать `value:3` как окончательный результат.

Новые вызовы `generator.next()` больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: `{done: true}`.

`function* f(…)` или `function *f(…)`?

Нет разницы, оба синтаксиса корректны.

Но обычно предпочтителен первый вариант, так как звёздочка относится к типу объявляемой сущности (`function*` – «функция-генератор»), а не к её названию, так что резонно расположить её у слова `function`.

Перебор генераторов

Как вы, наверное, уже догадались по наличию метода `next()`, генераторы являются перебираемыми объектами.

Возвращаемые ими значения можно перебирать через `for..of`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  return 3;
}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, затем 2
}
```

Выглядит гораздо красивее, чем использование `.next().value`, верно?

...Но обратите внимание: пример выше выводит значение `1`, затем `2`. Значение `3` выведено не будет!

Это из-за того, что перебор через `for..of` игнорирует последнее значение, при котором `done: true`. Поэтому, если мы хотим, чтобы были все значения при переборе через `for..of`, то надо возвращать их через `yield`:

```
function* generateSequence() {
  yield 1;
  yield 2;

}

let generator = generateSequence();

for(let value of generator) {
  alert(value); // 1, затем 2, затем 3
}
```

```
  yield 3;
```

Так как генераторы являются перебираемыми объектами, мы можем использовать всю связанную с ними функциональность, например оператор расширения `...`:

```
function* generateSequence() {
  yield 1;
  yield 2;
  yield 3;
}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3
```

В коде выше `...generateSequence()` превращает перебираемый объект-генератор в массив элементов (подробнее ознакомиться с оператором расширения можно в главе Остаточные параметры и оператор расширения)

Использование генераторов для перебираемых объектов

Некоторое время назад, в главе Перебираемые объекты, мы создали перебираемый объект `range`, который возвращает значения `from..to`.

Давайте вспомним код:

```
let range = {
  from: 1,
  to: 5,

  // for..of range вызывает этот метод один раз в самом начале
  [Symbol.iterator]() {
    // ...он возвращает перебираемый объект:
    // далее for..of работает только с этим объектом,
запрашивая следующие значения
    return {
      current: this.from,
      last: this.to,

      // next() вызывается при каждой итерации цикла for..of
      next() {
        // нужно вернуть значение как объект {done:.., value
:...}
        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
```

```
        } else {
          return { done: true };
        }
      }
    };
  }
};

// при переборе объекта range будут выведены числа от
range.from до range.to
alert([...range]); // 1,2,3,4,5
```

Мы можем использовать функцию-генератор для итерации, указав её в `Symbol.iterator`.

Вот тот же `range`, но с гораздо более компактным итератором:

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // краткая запись для
[Symbol.iterator]: function*()
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

alert( [...range] ); // 1,2,3,4,5
```

Это работает, потому что `range[Symbol.iterator]()` теперь возвращает генератор, и его методы – в точности то, что ожидает `for..of`:

- у него есть метод `.next()`
- который возвращает значения в виде `{value: ..., done: true/false}`

Это не совпадение, конечно. Генераторы были добавлены в язык JavaScript, в частности, с целью упростить создание перебираемых объектов.

Вариант с генератором намного короче, чем исходный вариант перебираемого `range`, и сохраняет те же функциональные возможности.

Генераторы могут генерировать бесконечно

В примерах выше мы генерировали конечные последовательности, но мы также можем сделать генератор, который будет возвращать значения бесконечно. Например, бесконечная последовательность псевдослучайных чисел.

Конечно, нам потребуется `break` (или `return`) в цикле `for..of` по такому генератору, иначе цикл будет продолжаться бесконечно, и скрипт «зависнет».

Композиция генераторов

Композиция генераторов – это особенная возможность генераторов, которая позволяет прозрачно «встраивать» генераторы друг в друга.

Например, у нас есть функция для генерации последовательности чисел:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}
```

Мы хотели бы использовать её при генерации более сложной последовательности:

- сначала цифры `0..9` (с кодами символов 48…57)
- за которыми следуют буквы в верхнем регистре `A..Z`

(коды символов 65…90)

за которыми следуют буквы алфавита `a..z` (коды символов 97…122)

Мы можем использовать такую последовательность для генерации паролей, выбирать символы из неё (может быть, ещё добавить символы пунктуации), но сначала её нужно сгенерировать.

В обычной функции, чтобы объединить результаты из нескольких других функций, мы вызываем их, сохраняем промежуточные результаты, а затем в конце их объединяем.

Для генераторов есть особый синтаксис `yield*`, который позволяет «вкладывать» генераторы один в другой (осуществлять их композицию).

Вот генератор с композицией:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generatePasswordCodes() {



}

let str = '';

for(let code of generatePasswordCodes()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9A..Za..z
```

```
  // 0..9
  yield* generateSequence(48, 57);

  // A..Z
  yield* generateSequence(65, 90);
```

```
  // a..z
  yield* generateSequence(97, 122);
```

Директива `yield*` *делегирует* выполнение другому генератору. Этот термин означает, что `yield* gen` перебирает генератор `gen` и прозрачно направляет его вывод наружу. Как если бы значения были сгенерированы внешним генератором.

Результат – такой же, как если бы мы встроили код из вложенных генераторов:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) yield i;
}

function* generateAlphaNum() {



}

let str = '';

for(let code of generateAlphaNum()) {
  str += String.fromCharCode(code);
}

alert(str); // 0..9a..zA..Z
  // yield* generateSequence(48, 57);
  for (let i = 48; i <= 57; i++) yield i;

  // yield* generateSequence(65, 90);
  for (let i = 65; i <= 90; i++) yield i;

  // yield* generateSequence(97, 122);
  for (let i = 97; i <= 122; i++) yield i;
```

Композиция генераторов – естественный способ вставлять вывод одного генератора в поток другого. Она не использует дополнительную память для хранения промежуточных результатов.

yield – дорога в обе стороны

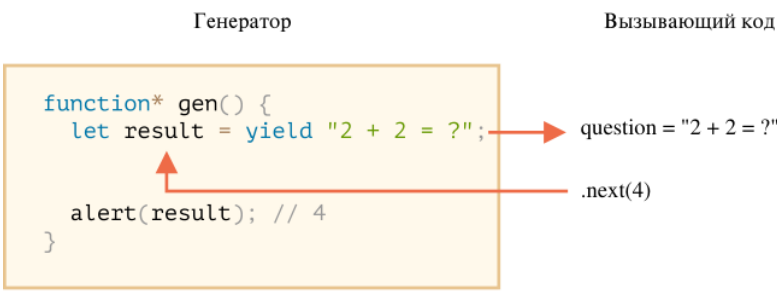
До этого момента генераторы сильно напоминали перебираемые объекты, со специальным синтаксисом для генерации значений. Но на самом деле они намного мощнее и гибче.

Всё дело в том, что `yield` – дорога в обе стороны: он не только возвращает результат наружу, но и может передавать значение извне в генератор.

Чтобы это сделать, нам нужно вызвать `generator.next(arg)` с аргументом. Этот аргумент становится результатом `yield`.

Продемонстрируем это на примере:

```
function* gen() {


  alert(result);
}

let generator = gen();

let question = generator.next().value; // <-- yield возвращает значение

generator.next(4); // --> передаём результат в генератор
  // Передаём вопрос во внешний код и ожидаем ответа
  let result = yield "2 + 2 = ?"; // (*)
```



1. Первый вызов `generator.next()` – всегда без аргумента, он начинает выполнение и возвращает результат первого `yield "2+2=?"`. На этой точке генератор приостанавливает выполнение.
2. Затем, как показано на картинке выше, результат `yield` переходит во внешний код в переменную `question`.
3. При `generator.next(4)` выполнение генератора возобновляется, а `4` выходит из присваивания как результат: `let result = 4`.

Обратите внимание, что внешний код не обязан немедленно вызывать `next(4)`. Ему может потребоваться время. Это не проблема, генератор подождёт.

Например:

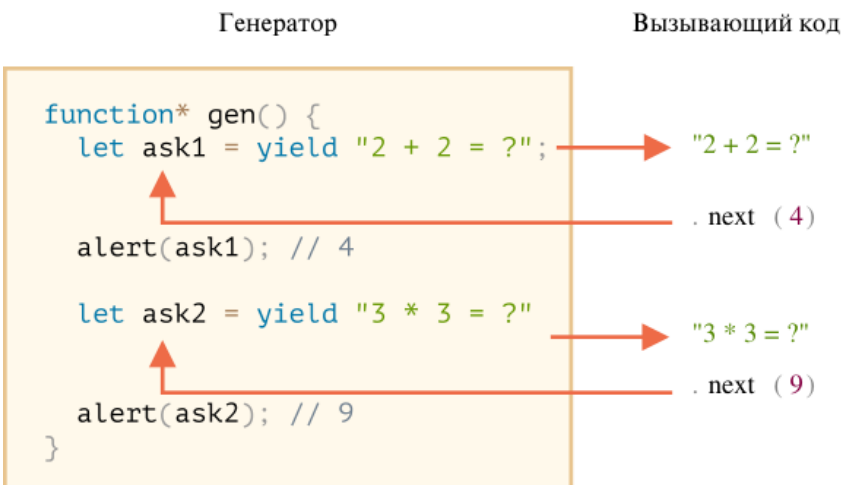
```
// возобновить генератор через некоторое время
setTimeout(() => generator.next(4), 1000);
```

Как видно, в отличие от обычных функций, генератор может обмениваться результатами с вызывающим кодом, передавая значения в `next/yield`.

Чтобы сделать происходящее более очевидным, вот ещё один пример с большим количеством вызовов:

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";

  alert(ask1); // 4

  let ask2 = yield "3 * 3 = ?"

  alert(ask2); // 9
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 * 3 = ?"

alert( generator.next(9).done ); // true
```

Картинка выполнения:



1. Первый `.next()` начинает выполнение… Оно доходит до первого `yield`.
2. Результат возвращается во внешний код.
3. Второй `.next(4)` передаёт `4` обратно в генератор как результат первого `yield` и возобновляет выполнение.
4. …Оно доходит до второго `yield`, который станет результатом `.next(4)`.
5. Третий `next(9)` передаёт `9` в генератор как

результат второго `yield` и возобновляет выполнение, которое завершается окончанием функции, так что `done: true`.

Получается такой «пинг-понг»: каждый `next(value)` передаёт в генератор значение, которое становится результатом текущего `yield`, возобновляет выполнение и получает выражение из следующего `yield`.

generator.throw

Как мы видели в примерах выше, внешний код может передавать значение в генератор как результат `yield`.

…Но можно передать не только результат, но и инициировать ошибку. Это естественно, так как ошибка является своего рода результатом.

Для того, чтобы передать ошибку в `yield`, нам нужно вызвать `generator.throw(err)`. В таком случае исключение `err` возникнет на строке с `yield`.

Например, здесь `yield "2 + 2 = ?"` приведёт к ошибке:

```
function* gen() {
  try {
    let result = yield "2 + 2 = ?"; // (1)

    alert("Выполнение программы не дойдёт до этой строки,
потому что выше возникнет исключение");
  } catch(e) {
    alert(e); // покажет ошибку
  }
}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("Ответ не найден в
```

```
моей базе данных")); // (2)
```

Ошибка, которая проброшена в генератор на строке `(2)`, приводит к исключению на строке `(1)` с `yield`. В примере выше `try..catch` перехватывает её и отображает.

Если мы не хотим перехватывать её, то она, как и любое обычное исключение, «вывалится» из генератора во внешний код.

Текущая строка вызывающего кода – это строка с `generator.throw`, отмечена `(2)`. Таким образом, мы можем отловить её во внешнем коде, как здесь:

```
function* generate() {
  let result = yield "2 + 2 = ?"; // Ошибка в этой строке
}

let generator = generate();

let question = generator.next().value;

try {
  generator.throw(new Error("Ответ не найден в моей базе данных"));
} catch(e) {
  alert(e); // покажет ошибку
}
```

Если же ошибка и там не перехвачена, то дальше – как обычно, она выпадает наружу и, если не перехвачена, «повалит» скрипт.

Итого

Генераторы создаются при помощи функций-генераторов `function* f(…) {…}`.

Внутри генераторов и только внутри них существует оператор `yield`.

Внешний код и генератор обмениваются промежуточными результатами посредством вызовов `next/yield`.

В современном JavaScript генераторы используются редко. Но иногда они оказываются полезными, потому что способность функции обмениваться данными с вызывающим кодом во время выполнения совершенно уникальна. И, конечно, для создания перебираемых объектов.

Также, в следующей главе мы будем изучать асинхронные генераторы, которые используются, чтобы читать потоки асинхронно сгенерированных данных (например, постранично загружаемые из сети) в цикле `for await ... of`.

В веб-программировании мы часто работаем с потоками данных, так что это ещё один важный случай использования.

Задачи

Псевдослучайный генератор

Есть много областей, где нам нужны случайные данные.

Одной из них является тестирование. Нам могут понадобиться случайные данные: текст, числа и т.д., чтобы хорошо всё проверить.

В JavaScript мы можем использовать `Math.random()`. Но если что-то пойдёт не так, то нам нужно будет

перезапустить тест, используя те же самые данные.

Для этого используются так называемые «сеяные псевдослучайные генераторы». Они получают «зерно», как первое значение, и затем генерируют следующее, используя формулу. Так что одно и то же зерно даёт одинаковую последовательность, и, следовательно, весь поток легко воспроизводим. Нам нужно только запомнить зерно, чтобы воспроизвести последовательность.

Пример такой формулы, которая генерирует более-менее равномерно распределённые значения:

```
next = previous * 16807 % 2147483647
```

Если мы используем `1` как зерно, то значения будут:

1. `16807`
2. `282475249`
3. `1622650073`
4. …и так далее…

Задачей является создать функцию-генератор `pseudoRandom(seed)`, которая получает `seed` и создаёт генератор с указанной формулой.

Пример использования:

```
let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

Открыть песочницу с тестами для задачи.

К решению

Асинхронные итераторы и генераторы

Асинхронные итераторы позволяют перебирать данные, поступающие асинхронно. Например, когда мы загружаем что-то по частям по сети. Асинхронные генераторы делают такой перебор ещё удобнее.

Давайте сначала рассмотрим простой пример, чтобы понять синтаксис, а затем – реальный практический.

Асинхронные итераторы

Асинхронные итераторы похожи на обычные итераторы, но имеют некоторые синтаксические отличия.

«Обычный» перебираемый объект, как подробно рассказано в главе Перебираемые объекты, выглядит примерно так:

```
let range = {
  from: 1,
  to: 5,

  // for..of вызывает этот метод один раз в самом начале

    // ...возвращает объект-итератор:
    // далее for..of работает только с этим объектом,
запрашивая следующее значение вызовом next()
    return {
      current: this.from,
      last: this.to,

      // next() вызывается на каждой итерации цикла for..of

        if (this.current <= this.last) {
```

```
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
 }
};

for(let value of range) {
  alert(value); // 1, потом 2, потом 3, потом 4, потом 5
}
```

```
  [Symbol.iterator]() {
      next() { // (2)
        // должен возвращать значение в виде
объекта {done:.., value :...}
```

Если нужно, пожалуйста, ознакомьтесь с главой про итераторы, где обычные итераторы разбираются подробно.

Чтобы сделать объект итерируемым асинхронно:

1. Используется `Symbol.asyncIterator` вместо `Symbol.iterator`.
2. `next()` должен возвращать промис.
3. Чтобы перебрать такой объект, используется цикл `for await (let item of iterable)`.

Давайте создадим итерируемый объект `range`, как и в предыдущем примере, но теперь он будет возвращать значения асинхронно, по одному в секунду:

```
let range = {
  from: 1,
  to: 5,

  // for await..of вызывает этот метод один раз в самом начале

    // ...возвращает объект-итератор:
    // далее for await..of работает только с этим объектом,
    // запрашивая у него следующие значения вызовом next()
    return {
```

```
      current: this.from,
      last: this.to,

      // next() вызывается на каждой итерации цикла for
await..of


        // можно использовать await внутри для асинхронности:
        await new Promise(resolve => setTimeout(resolve,
1000)); // (3)

        if (this.current <= this.last) {
          return { done: false, value: this.current++ };
        } else {
          return { done: true };
        }
      }
    };
  }
};

(async () => {



})()
  [Symbol.asyncIterator]() { // (1)
      async next() { // (2)
        // должен возвращать значение как
объект {done:.., value :...}
        // (автоматически оборачивается в
промис с помощью async)
  for await (let value of range) { // (4)
    alert(value); // 1,2,3,4,5
  }
```

Как видим, структура похожа на обычные итераторы:

1. Чтобы сделать объект асинхронно итерируемым, он должен иметь метод `Symbol.asyncIterator` (1).
2. Этот метод должен возвращать объект с методом `next()`, который в свою очередь возвращает промис (2).
3. Метод `next()` не обязательно должен быть `async`, он может быть обычным методом, возвращающим

промис, но `async` позволяет использовать `await`, так что это удобно. Здесь мы просто делаем паузу на одну секунду `(3)`.

4. Для итерации мы используем `for await (let value of range) (4)`, добавляя «await» после «for». Он вызовет `range[Symbol.asyncIterator]()` один раз, а затем его метод `next()` для получения значений.

Вот небольшая шпаргалка:

	Итераторы	Асинхронные итераторы
Метод для создания итерируемого объекта	`Symbol.iterator`	`Symbol.asyncItera`
`next()` возвращает	любое значение	промис
для цикла используйте	`for..of`	`for await..of`

Оператор расширения `...` не работает асинхронно

Функции, которые требуют обычных синхронных итераторов, не работают с асинхронными.

Например, оператор расширения (три точки `...`) не будет работать:

```
alert( [...range] ); // Ошибка, нет Symbol.iterator
```

Это естественно, так как он ожидает `Symbol.iterator`, как и `for..of` без `await`. Ему не подходит `Symbol.asyncIterator`.

Асинхронные генераторы

Как мы уже знаем, в JavaScript есть генераторы, и они являются перебираемыми.

Давайте вспомним генератор последовательности из главы Генераторы. Он генерирует последовательность значений от `start` до `end`:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {
    yield i;
  }
}

for(let value of generateSequence(1, 5)) {
  alert(value); // 1, потом 2, потом 3, потом 4, потом 5
}
```

В обычных генераторах мы не можем использовать `await`. Все значения должны поступать синхронно: в `for..of` нет места для задержки, это синхронная конструкция.

Но что если нам нужно использовать `await` в теле генератора? Для выполнения сетевых запросов, например.

Нет проблем, просто добавьте в начале `async`, например, вот так:

```
function* generateSequence(start, end) {
  for (let i = start; i <= end; i++) {


    yield i;
```

```
  }

}

(async () => {

  let generator = generateSequence(1, 5);
  for (let value of generator) {
    alert(value); // 1, потом 2, потом 3, потом 4, потом 5
  }

})();
async
    // ура, можно использовать await!
    await new Promise(resolve =>
setTimeout(resolve, 1000));
await
```

Теперь у нас есть асинхронный генератор, который можно перебирать с помощью `for await ... of`.

Это действительно очень просто. Мы добавляем ключевое слово `async`, и внутри генератора теперь можно использовать `await`, а также промисы и другие асинхронные функции.

С технической точки зрения, ещё одно отличие асинхронного генератора заключается в том, что его метод `generator.next()` теперь тоже асинхронный и возвращает промисы.

Из обычного генератора мы можем получить значения при помощи `result = generator.next()`. Для асинхронного нужно добавить `await`, вот так:

```
result = await generator.next(); // result = {value: ...,
done: true/false}
```

Асинхронно перебираемые объекты

Как мы уже знаем, чтобы сделать объект перебираемым,

нужно добавить к нему `Symbol.iterator`.

```
let range = {
  from: 1,
  to: 5,

}
   [Symbol.iterator]() {
     return <объект с next, чтобы сделать range
перебираемым>
   }
```

Обычная практика для `Symbol.iterator` – возвращать генератор, а не простой объект с `next`, как в предыдущем примере.

Давайте вспомним пример из главы Генераторы:

```
let range = {
  from: 1,
  to: 5,

  *[Symbol.iterator]() { // сокращение для [Symbol.iterator]:
function*()
    for(let value = this.from; value <= this.to; value++) {
      yield value;
    }
  }
};

for(let value of range) {
  alert(value); // 1, потом 2, потом 3, потом 4, потом 5
}
```

Здесь созданный объект `range` является перебираемым, а генератор `*[Symbol.iterator]` реализует логику для перечисления значений.

Если хотим добавить асинхронные действия в генератор, нужно заменить `Symbol.iterator` на асинхронный `Symbol.asyncIterator`:

```
let range = {
  from: 1,
  to: 5,


    for(let value = this.from; value <= this.to; value++) {

      // пауза между значениями, ожидание
      await new Promise(resolve => setTimeout(resolve, 1000));

      yield value;
    }
  }
};

(async () => {

  for  (let value of range) {
    alert(value); // 1, потом 2, потом 3, потом 4, потом 5
  }

})();
  async *[Symbol.asyncIterator]() { // то же,
что и [Symbol.asyncIterator]: async
function*()
await
```

Теперь значения поступают с задержкой в одну секунду между ними.

Пример из реальной практики

До сих пор мы видели простые примеры, чтобы просто получить базовое представление. Теперь давайте рассмотрим реальную ситуацию.

Есть много онлайн-сервисов, которые предоставляют данные постранично. Например, когда нам нужен список пользователей, запрос возвращает предопределённое количество (например, 100) пользователей – «одну страницу», и URL следующей страницы.

Этот подход очень распространён, и речь не только о

пользователях, а о чём угодно. Например, GitHub позволяет получать коммиты таким образом, с разбивкой по страницам:

> Нужно сделать запрос на URL в виде `https://api.github.com/repos/<repo>/commits`.
>
> В ответ придёт JSON с 30 коммитами, а также со ссылкой на следующую страницу в заголовке `Link`.
>
> Затем можно использовать эту ссылку для следующего запроса, чтобы получить дополнительную порцию коммитов, и так далее.

Но нам бы, конечно же, хотелось вместо этого сложного взаимодействия иметь просто объект с коммитами, которые можно перебирать, вот так:

```
let repo = 'javascript-tutorial/en.javascript.info'; // репозиторий на GitHub, откуда брать коммиты

for await (let commit of fetchCommits(repo)) {
  // обработка коммитов
}
```

Мы бы хотели сделать функцию `fetchCommits(repo)`, которая будет получать коммиты, делая запросы всякий раз, когда это необходимо. И пусть она сама разбирается со всем, что касается нумерации страниц, для нас это будет просто `for await..of`.

С асинхронными генераторами это довольно легко реализовать:

```
async function* fetchCommits(repo) {
  let url = `https://api.github.com/repos/${repo}/commits`;

  while (url) {
    const response = await fetch(url, { // (1)
      headers: {'User-Agent': 'Our script'}, // GitHub требует заголовок user-agent
    });
```

```
    const body = await response.json(); // (2) ответ в формате
JSON (массив коммитов)

    // (3) Ссылка на следующую страницу находится в
заголовках, извлекаем её
    let nextPage = response.headers.get('Link').match(/
<(.*?)>; rel="next"/);
    nextPage = nextPage && nextPage[1];

    url = nextPage;

    for(let commit of body) { // (4) вернуть коммиты один за
другим, до окончания страницы
      yield commit;
    }
  }
}
```

1. Мы используем метод fetch браузера для загрузки с удалённого URL. Он позволяет при необходимости добавлять авторизацию и другие заголовки, здесь GitHub требует `User-Agent`.
2. Результат `fetch` обрабатывается как JSON, это опять-таки метод, присущий `fetch`.
3. Нужно получить URL следующей страницы из заголовка ответа `Link`. Он имеет специальный формат, поэтому мы используем регулярное выражение. URL следующей страницы может выглядеть как `https://api.github.com/repositories/93253246/commits?page=2`, он генерируется самим GitHub.
4. Затем мы выдаём все полученные коммиты, а когда они закончатся – сработает следующая итерация `while(url)`, которая сделает ещё один запрос.

Пример использования (показывает авторов коммитов в консоли):

```
(async () => {

  let count = 0;

  for await (const commit of fetchCommits('javascript-
```

```
tutorial/en.javascript.info')) {

    console.log(commit.author.login);

    if (++count == 100) { // остановимся на 100 коммитах
      break;
    }
  }

})();
```

Это именно то, что мы хотели. Внутренняя механика постраничных запросов снаружи не видна. Для нас это просто асинхронный генератор, который возвращает коммиты.

Итого

Обычные итераторы и генераторы прекрасно работают с данными, которые не требуют времени для их создания или получения.

Когда мы ожидаем, что данные будут поступать асинхронно, с задержками, можно использовать их асинхронные аналоги и `for await..of` вместо`for..of`.

Синтаксические различия между асинхронными и обычными итераторами:

	Перебираемый объект	**Асинхронно перебираемый**
Метод для получения итератора	`Symbol.iterator`	`Symbol.asyncItera`
`next()` возвращает	`{value:…, done: true/false}`	промис, который завершается с `{value:…, done: true/`

	`false}`

Синтаксические различия между асинхронными и обычными генераторами:

	Генераторы	**Асинхронные генераторы**
Объявление	`function*`	`async function*`
`generator.next()` возвращает	`{value:…, done: true/false}`	промис, который завершается с `{value:…, done: true/false}`

В веб-разработке мы часто встречаемся с потоками данных, когда они поступают по частям. Например, загрузка или выгрузка большого файла.

Мы можем использовать асинхронные генераторы для обработки таких данных. Также заметим, что в некоторых окружениях, например, браузерах, есть и другое API, называемое Streams (потоки), который предоставляет специальные интерфейсы для работы с такими потоками данных, их преобразования и передачи из одного потока в другой (например, загрузка из одного источника и сразу отправка в другое место).

Модули

Модули, введение

По мере роста нашего приложения, мы обычно хотим разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями.

Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости.

Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей.

Например:

- AMD – одна из самых старых модульных систем, изначально реализована библиотекой require.js.
- CommonJS – модульная система, созданная для сервера Node.js.
- UMD – ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

Теперь все они постепенно становятся частью истории, хотя их и можно найти в старых скриптах.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js. Далее мы будем изучать именно её.

Что такое модуль?

Модуль – это просто файл. Один скрипт – это один модуль.

Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

> `export` отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
>
> `import` позволяет импортировать функциональность из других модулей.

Например, если у нас есть файл `sayHi.js`, который экспортирует функцию:

```
// 📁 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

…Тогда другой файл может импортировать её и использовать:

```
// 📁 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

Директива `import` загружает модуль по пути `./sayHi.js` относительно текущего файла и записывает экспортированную функцию `sayHi` в соответствующую переменную.

Давайте запустим пример в браузере.

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута `<script type="module">`.

Вот так:

https://plnkr.co/edit/tI0csVxfZ3Wu2yiO?p=preview

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.

Основные возможности модулей

Чем отличаются модули от «обычных» скриптов?

Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

Всегда «use strict»

В модулях всегда используется режим `use strict`. Например, присваивание к необъявленной переменной вызовет ошибку.

```
<script type="module">
  a = 5; // ошибка
</script>
```

Своя область видимости переменных

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и `hello.js` пытается использовать переменную `user`, объявленную в `user.js`. В итоге ошибка:

https://plnkr.co/edit/NU3BTQyPEYPHlwsD?p=preview

Модули должны экспортировать функциональность, предназначенную для использования извне. А другие модули могут её импортировать.

Так что нам надо импортировать `user.js` в `hello.js` и взять из него нужную функциональность, вместо того чтобы полагаться на глобальные переменные.

Правильный вариант:

https://plnkr.co/edit/t771Q4vDngvDTsOE?p=preview

В браузере также существует независимая область видимости для каждого скрипта `<script type="module">`:

```
<script type="module">
  // Переменная доступна только в этом модуле
  let user = "John";
</script>

<script type="module">

</script>
  alert(user); // Error: user is not defined
```

Если нам нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту `window`, тогда получить значение переменной можно обратившись к `window.user`. Но это должно быть исключением, требующим веской причины.

Код в модуле выполняется только один раз при импорте

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

Это очень важно для понимания работы модулей. Давайте посмотрим примеры.

Во-первых, если при запуске модуля возникают побочные эффекты, например выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

```
// 📄 alert.js
alert("Модуль выполнен!");


// Импорт одного и того же модуля в разных файлах

// 📄 1.js
import `./alert.js`; // Модуль выполнен!

// 📄 2.js
import `./alert.js`; // (ничего не покажет)
```

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если мы хотим, чтобы что-то можно было использовать много раз, то экспортируем это.

Теперь более продвинутый пример.

Давайте представим, что модуль экспортирует объект:

```
// 📄 admin.js
export let admin = {
  name: "John"
};
```

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект `admin` будет создан и в дальнейшем будет передан всем импортёрам.

Все импортёры получат один-единственный объект `admin`:

```
// 📄 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 📄 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Оба файла, 1.js и 2.js, импортируют один и
тот же объект
// Изменения, сделанные в 1.js, будут видны в
2.js
```

Ещё раз заметим – модуль выполняется только один раз. Генерируется экспорт и после передаётся всем импортёрам, поэтому, если что-то изменится в объекте `admin`, то другие модули тоже увидят эти изменения.

Такое поведение позволяет *конфигурировать* модули при первом импорте. Мы можем установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль `admin.js` предоставляет определённую функциональность, но ожидает передачи учётных данных в объект `admin` извне:

```
// 📄 admin.js
export let admin = { };

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}
```

В `init.js`, первом скрипте нашего приложения, мы установим `admin.name`. Тогда все это увидят, включая вызовы, сделанные из самого `admin.js`:

```
// 📄 init.js
import {admin} from './admin.js';
admin.name = "Pete";
```

Другой модуль тоже увидит `admin.name`:

```
// other.js
import {admin, sayHi} from './admin.js';

alert(admin.name); //

sayHi(); // Ready to serve, !
Pete
Pete
```

import.meta

Объект `import.meta` содержит информацию о текущем модуле.

Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

```
<script type="module">
  alert(import.meta.url); // ссылка на html страницу для встроенного скрипта
</script>
```

В модуле «this» не определён

Это незначительная особенность, но для полноты картины нам нужно упомянуть об этом.

В модуле на верхнем уровне `this` не определён (undefined).

Сравним с не-модульными скриптами, там `this` – глобальный объект:

```
<script>
  alert(this); // window
</script>

<script type="module">
  alert(this); // undefined
</script>
```

Особенности в браузерах

Есть и несколько других, именно браузерных особенностей скриптов с `type="module"` по сравнению с обычными скриптами.

Если вы читаете материал в первый раз или, если не собираетесь использовать модули в браузерах, то сейчас можете пропустить эту секцию.

Модули являются отложенными (deferred)

Модули *всегда* выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом `defer` (описан в главе Скрипты: async, defer). Это верно и для внешних и встроенных скриптов-модулей.

Другими словами:

- загрузка внешних модулей, таких как `<script type="module" src="...">`, не блокирует обработку HTML.
- модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
- сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними.

Например:

```
<script type="module">

  // так как модули являются отложенными, то скрипт начнёт
выполнятся только после полной загрузки страницы
</script>

Сравните с обычным скриптом ниже:
```

```
<script>

  // обычные скрипты запускаются сразу, не дожидаясь полной
загрузки страницы
</script>

<button id="button">Кнопка</button>
  alert(typeof button); // object: скрипт
может 'видеть' кнопку под ним
  alert(typeof button); // Ошибка: кнопка не
определена, скрипт не видит элементы под ним
```

Пожалуйста, обратите внимание: второй скрипт выполнится раньше, чем первый! Поэтому мы увидим сначала `undefined`, а потом `object`.

Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта мы видим первым.

При использовании модулей нам стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Нам следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

Атрибут async работает во встроенных скриптах

Для не-модульных скриптов атрибут `async` работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут `async` работает на любых скриптах.

Например, в скрипте ниже есть `async`, поэтому он выполнится сразу после загрузки, не ожидая других скриптов.

Скрипт выполнит импорт (загрузит `./analytics.js`) и

сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются.

Это очень полезно, когда модуль ни с чем не связан, например для счётчиков, рекламы, обработчиков событий.

```
<!-- загружаются зависимости (analytics.js) и скрипт
запускается -->
<!-- модуль не ожидает загрузки документа или других тэгов
<script> -->
<script  type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
async
```

Внешние скрипты

Внешние скрипты с атрибутом `type="module"` имеют два отличия:

1. Внешние скрипты с одинаковым атрибутом `src` запускаются только один раз:

   ```
   <!-- скрипт my.js загрузится и будет выполнен только один
   раз -->
   <script type="module" src="my.js"></script>
   <script type="module" src="my.js"></script>
   ```

2. Внешний скрипт, который загружается с другого домена, требует указания заголовков CORS. Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок `Access-Control-Allow-Origin` означающий, что загрузка скрипта разрешена.

   ```
   <!-- another-site.com должен указать заголовок Access-
   Control-Allow-Origin -->
   <!-- иначе, скрипт не выполнится -->
   <script type="module" src=""></script>
   ```

```
http://another-site.com/their.js
```

Это обеспечивает лучшую безопасность по умолчанию.

Не допускаются «голые» модули

В браузере `import` должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в `import`.

Например, этот `import` неправильный:

```
import {sayHi} from 'sayHi'; // Ошибка, "голый" модуль
// путь должен быть, например './sayHi.js' или абсолютный
```

Другие окружения, например Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Совместимость, «nomodule»

Старые браузеры не понимают атрибут `type="module"`. Скрипты с неизвестным атрибутом `type` просто игнорируются. Мы можем сделать для них «резервный» скрипт при помощи атрибута `nomodule`:

```
<script type="module">
  alert("Работает в современных браузерах");
</script>

<script nomodule>
  alert("Современные браузеры понимают оба атрибута - и
type=module, и nomodule, поэтому пропускают этот тег script")
  alert("Старые браузеры игнорируют скрипты с неизвестным
атрибутом type=module, но выполняют этот.");
</script>
```

Инструменты сборки

В реальной жизни модули в браузерах редко используются в «сыром» виде. Обычно, мы объединяем модули вместе, используя специальный инструмент, например Webpack и после выкладываем код на рабочий сервер.

Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «своё», например CSS/HTML-модули.

Сборщик делает следующее:

1. Берёт «основной» модуль, который мы собираемся поместить в `<script type="module">` в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее)
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный `import` функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.
4. В процессе могут происходить и другие трансформации и оптимизации кода:

    Недостижимый код удаляется.

    Неиспользуемые экспорты удаляются («tree-shaking»).

    Специфические операторы для разработки, такие как `console` и `debugger`, удаляются.

    Современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью, например, с помощью Babel.

    Полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если мы используем инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют `import/export` на свои вызовы. Поэтому итоговую сборку можно подключать и без атрибута `type="module"`, как обычный скрипт:

```
<!-- Предположим, что мы собрали bundle.js, используя например утилиту Webpack -->
<script src="bundle.js"></script>
```

Хотя и «как есть» модули тоже можно использовать, а сборщик настроить позже при необходимости.

Итого

Подводя итог, основные понятия:

1. Модуль – это файл. Чтобы работал `import/export`, нужно для браузеров указывать атрибут `<script type="module">`. У модулей есть ряд особенностей:
    - Отложенное (deferred) выполнение по умолчанию.
    - Атрибут async работает во встроенных скриптах.
    - Для загрузки внешних модулей с другого источника, он должен ставить заголовки CORS.
    - Дублирующиеся внешние скрипты игнорируются.
6. У модулей есть своя область видимости, обмениваться функциональностью можно через `import/export`.
7. В модулях всегда включена директива `use strict`.
8. Код в модулях выполняется только один раз. Экспортируемая функциональность создаётся один раз и передаётся всем импортёрам.

Когда мы используем модули, каждый модуль реализует свою функциональность и экспортирует её. Затем мы используем `import`, чтобы напрямую импортировать её

туда, куда необходимо. Браузер загружает и анализирует скрипты автоматически.

В реальной жизни часто используется сборщик Webpack, чтобы объединить модули: для производительности и других «плюшек».

В следующей главе мы увидим больше примеров и вариантов импорта/экспорта.

Экспорт и импорт

Директивы экспорт и импорт имеют несколько вариантов вызова.

В предыдущей главе мы видели простое использование, давайте теперь посмотрим больше примеров.

Экспорт до объявления

Мы можем пометить любое объявление как экспортируемое, разместив `export` перед ним, будь то переменная, функция или класс.

Например, все следующие экспорты допустимы:

```
// экспорт массива
 let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec'];

// экспорт константы
 const MODULES_BECAME_STANDARD_YEAR = 2015;

// экспорт класса
 class User {
  constructor(name) {
    this.name = name;
  }
}
export
```

```
export
export
```

Не ставится точка с запятой после экспорта класса/функции

Обратите внимание, что `export` перед классом или функцией не делает их функциональным выражением. Это всё также объявление функции, хотя и экспортируемое.

Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов.

Поэтому в конце `export class` и `export function` не нужна точка с запятой:

```
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
 // без ; в конце
```

Экспорт отдельно от объявления

Также можно написать `export` отдельно.

Здесь мы сначала объявляем, а затем экспортируем:

```
// 📄 say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}
```

```
export {sayHi, sayBye}; // список
экспортируемых переменных
```

…Или, технически, мы также можем расположить `export` выше функций.

Импорт *

Обычно мы располагаем список того, что хотим импортировать, в фигурных скобках `import {...}`, например вот так:

```
// 📄 main.js

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
import {sayHi, sayBye} from './say.js';
```

Но если импортировать нужно много чего, мы можем импортировать всё сразу в виде объекта, используя `import * as <obj>`. Например:

```
// 📄 main.js

say.sayHi('John');
say.sayBye('John');
import * as say from './say.js';
```

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего, зачем нам вообще может понадобиться явно перечислять список того, что нужно импортировать?

Для этого есть несколько причин.

1. Современные инструменты сборки (webpack и другие)

собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

Предположим, мы добавили в наш проект стороннюю библиотеку `say.js` с множеством функций:

```
// 📁 say.js
export function sayHi() { ... }
export function sayBye() { ... }
export function becomeSilent() { ... }
```

Теперь, если из этой библиотеки в проекте мы используем только одну функцию:

```
// 📁 main.js
import {sayHi} from './say.js';
```

…Тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

2. Явно перечисляя то, что хотим импортировать, мы получаем более короткие имена функций: `sayHi()` вместо `say.sayHi()`.
3. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

Импорт «как»

Мы также можем использовать `as`, чтобы импортировать под другими именами.

Например, для краткости импортируем `sayHi` в локальную переменную `hi`, а `sayBye` импортируем как

bye:

```
// 📄 main.js

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

```
import {sayHi as hi, sayBye as bye} from './
say.js';
```

Экспортировать «как»

Аналогичный синтаксис существует и для `export`.

Давайте экспортируем функции, как `hi` и `bye`:

```
// 📄 say.js
...
export {sayHi as hi, sayBye as bye};
```

Теперь `hi` и `bye` – официальные имена для внешнего кода, их нужно использовать при импорте:

```
// 📄 main.js
import * as say from './say.js';

say.('John'); // Hello, John!
say.('John'); // Bye, John!
```

```
hi
bye
```

Экспорт по умолчанию

На практике модули встречаются в основном одного из двух типов:

1. Модуль, содержащий библиотеку или набор функций, как `say.js` выше.

2. Модуль, который объявляет что-то одно, например модуль `user.js` экспортирует только `class User`.

По большей части, удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле.

Естественно, требуется много файлов, если для всего делать отдельный модуль, но это не проблема. Так даже удобнее: навигация по проекту становится проще, особенно, если у файлов хорошие имена, и они структурированы по папкам.

Модули предоставляют специальный синтаксис `export default` («экспорт по умолчанию») для второго подхода.

Ставим `export default` перед тем, что нужно экспортировать:

```
// 📄 user.js
export  class User { // просто добавьте "default"
  constructor(name) {
    this.name = name;
  }
}
```
```
default
```

Заметим, в файле может быть не более одного `export default`.

…И потом импортируем без фигурных скобок:

```
// 📄 main.js
import  from './user.js'; // не {User}, просто User

new User('John');
```
```
User
```

Импорты без фигурных скобок выглядят красивее. Обычная ошибка начинающих: забывать про фигурные скобки. Запомним: фигурные скобки необходимы в случае

именованных экспортов, для `export default` они не нужны.

Именованный экспорт	Экспорт по умолчанию
`export class User {...}`	`export default class User {...}`
`import {User} from ...`	`import User from ...`

Технически в одном модуле может быть как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают. То есть, в модуле находятся либо именованные экспорты, либо один экспорт по умолчанию.

Так как в файле может быть максимум один `export default`, то экспортируемая сущность не обязана иметь имя.

Например, всё это – полностью корректные экспорты по умолчанию:

```
export default class { // у класса нет имени
  constructor() { ... }
}


export default function(user) { // у функции нет имени
  alert(`Hello, ${user}!`);
}


// экспортируем значение, не создавая переменную
export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec'];
```

Это нормально, потому что может быть только один `export default` на файл, так что `import` без фигурных скобок всегда знает, что импортировать.

Без `default` такой экспорт выдал бы ошибку:

```
export class { // Ошибка! (необходимо имя, если это не экспорт по умолчанию)
  constructor() {}
}
```

Имя «default»

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется `default`.

Например, чтобы экспортировать функцию отдельно от её объявления:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// то же самое, как если бы мы добавили "export default" перед функцией
export {sayHi as default};
```

Или, ещё ситуация, давайте представим следующее: модуль `user.js` экспортирует одну сущность «по умолчанию» и несколько именованных (редкий, но возможный случай):

```
// 📁 user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

```
// 📄 main.js
import {, sayHi} from './user.js';

new User('John');
default
 as User
```

И, наконец, если мы импортируем всё как объект `import *`, тогда его свойство `default` – как раз и будет экспортом по умолчанию:

```
// 📄 main.js
import * as user from './user.js';

let User = user.default; // экспорт по умолчанию
new User('John');
```

Довод против экспортов по умолчанию

Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, говорит нам, что именно экспортируется.

Именованные экспорты вынуждают нас использовать правильное имя при импорте:

```
import {User} from './user.js';
// import {MyUser} не сработает, должно быть именно имя {User}
```

…В то время как для экспорта по умолчанию мы выбираем любое имя при импорте:

```
import User from './user.js'; // сработает
import MyUser from './user.js'; // тоже сработает
// можно импортировать с любым именем, и это будет работать
```

Так что члены команды могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо.

Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных

должны соответствовать именам файлов. Вот так:

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
...
```

Тем не менее, в некоторых командах это считают серьёзным доводом против экспортов по умолчанию и предпочитают использовать именованные экспорты везде. Даже если экспортируется только одна вещь, она всё равно экспортируется с именем, без использования `default`.

Это также немного упрощает реэкспорт (смотрите ниже).

Реэкспорт

Синтаксис «реэкспорта» `export ... from ...` позволяет импортировать что-то и тут же экспортировать, возможно под другим именем, вот так:

```
export {sayHi} from './say.js'; // реэкспортировать sayHi

export {default as User} from './user.js'; // реэкспортировать
default
```

Зачем это нужно? Рассмотрим практический пример использования.

Представим, что мы пишем «пакет»: папку со множеством модулей, из которой часть функциональности экспортируется наружу (инструменты вроде NPM позволяют нам публиковать и распространять такие пакеты), а многие модули – просто вспомогательные, для внутреннего использования в других модулях пакета.

Структура файлов может быть такой:

```
auth/
    index.js
    user.js
    helpers.js
    tests/
        login.js
    providers/
        github.js
        facebook.js
        ...
```

Мы бы хотели сделать функциональность нашего пакета доступной через единую точку входа: «главный файл» `auth/index.js`. Чтобы можно было использовать её следующим образом:

```
import {login, logout} from 'auth/index.js'
```

Идея в том, что внешние разработчики, которые будут использовать наш пакет, не должны разбираться с его внутренней структурой, рыться в файлах внутри нашего пакета. Всё, что нужно, мы экспортируем в `auth/index.js`, а остальное скрываем от любопытных взглядов.

Так как нужная функциональность может быть разбросана по модулям нашего пакета, мы можем импортировать их в `auth/index.js` и тут же экспортировать наружу.

```
// 📄 auth/index.js

// импортировать login/logout и тут же экспортировать
import {login, logout} from './helpers.js';
export {login, logout};

// импортировать экспорт по умолчанию как User и тут же
экспортировать
import User from './user.js';
export {User};
...
```

Теперь пользователи нашего пакета могут писать `import {login} from "auth/index.js"`.

Запись `export ... from ...` – это просто более короткий вариант такого импорта-экспорта:

```
// 📄 auth/index.js

// импортировать login/logout и тут же экспортировать
export {login, logout} from './helpers.js';

// импортировать экспорт по умолчанию как User и тут же
экспортировать
export {default as User} from './user.js';
...
```

Реэкспорт экспорта по умолчанию

При реэкспорте экспорт по умолчанию нужно обрабатывать особым образом.

Например, у нас есть `user.js`, из которого мы хотим реэкспортировать класс `User`:

```
// 📄 user.js
export default class User {
  // ...
}
```

1. `export User from './user.js'` не будет работать. Казалось бы, что такого? Но возникнет синтаксическая ошибка!

   Чтобы реэкспортировать экспорт по умолчанию, мы должны написать `export {default as User}`, как в примере выше. Такая вот особенность синтаксиса.

2. `export * from './user.js'` реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

   Если мы хотим реэкспортировать и именованные экспорты и экспорт по умолчанию, то понадобятся две инструкции:

```
export * from './user.js'; // для реэкспорта именованных
экспортов
export {default} from './user.js'; // для реэкспорта по
умолчанию
```

Такое особое поведение реэкспорта с экспортом по умолчанию – одна из причин того, почему некоторые разработчики их не любят.

Итого

Вот все варианты `export`, которые мы разобрали в этой и предыдущей главах.

Вы можете проверить себя, читая их и вспоминая, что они означают:

Перед объявлением класса/функции/...:

```
export [default] class/function/variable
...
```

Отдельный экспорт:

```
export {x [as y], ...}.
```

Реэкспорт:

```
export {x [as y], ...} from "module"
```

`export * from "module"` (не реэкспортирует `export default`).

`export {default [as y]} from "module"` (реэкспортирует только `export default`).

Импорт:

Именованные экспорты из модуля:

```
import {x [as y], ...} from "module"
```

Импорт по умолчанию:

```
import x from "module"
```

```
import {default as x} from "module"
```

Всё сразу:

```
import * as obj from "module"
```

Только подключить модуль (его код запустится), но не присваивать его переменной:

```
import "module"
```

Мы можем поставить `import/export` в начало или в конец скрипта, это не имеет значения.

То есть, технически, такая запись вполне корректна:

```
sayHi();

// ...

import {sayHi} from './say.js'; // импорт в конце файла
```

На практике импорты, чаще всего, располагаются в начале файла. Но это только для большего удобства.

Обратите внимание, что инструкции import/export не работают внутри `{...}`.

Условный импорт, такой как ниже, работать не будет:

```
if (something) {
  import {sayHi} from "./say.js"; // Ошибка: импорт должен быть на верхнем уровне
}
```

…Но что, если нам в самом деле нужно импортировать что-либо в зависимости от условий? Или в определённое время? Например, загрузить модуль, только когда он станет нужен?

Мы рассмотрим динамические импорты в следующей главе.

Динамические импорты

Инструкции экспорта и импорта, которые мы рассматривали в предыдущей главе, называются «статическими». Синтаксис у них весьма простой и строгий.

Во-первых, мы не можем динамически задавать никакие из параметров `import`.

Путь к модулю должен быть строковым примитивом и не может быть вызовом функции. Вот так работать не будет:

```
import ... from ; // Ошибка, должна быть строка
getModuleName
()
```

Во-вторых, мы не можем делать импорт в зависимости от условий или в процессе выполнения.

```
if(...) {
  import ...; // Ошибка, запрещено
}

{
  import ...; // Ошибка, мы не можем ставить импорт в блок
}
```

Всё это следствие того, что цель директив `import/export` – задать костяк структуры кода. Благодаря им она может быть проанализирована, модули могут быть собраны в один файл специальными инструментами, а неиспользуемые экспорты удалены. Это возможно только благодаря тому, что всё статично.

Но как мы можем импортировать модуль динамически, по запросу?

Выражение import()

Выражение `import(module)` загружает модуль и возвращает промис, результатом которого становится объект модуля, содержащий все его экспорты.

Использовать его мы можем динамически в любом месте кода, например, так:

```
let modulePath = prompt("Какой модуль загружать?");

import(modulePath)
  .then(obj => <объект модуля>)
  .catch(err => <ошибка загрузки, например если нет такого
модуля>)
```

Или если внутри асинхронной функции, то можно `let module = await import(modulePath)`.

Например, если у нас есть такой модуль `say.js`:

```
// 📁 say.js
export function hi() {
  alert(`Привет`);
}

export function bye() {
  alert(`Пока`);
}
```

…То динамический импорт может выглядеть так:

```
let {hi, bye} = await import('./say.js');

hi();
bye();
```

А если в `say.js` указан экспорт по умолчанию:

```
// 📁 say.js
export default function() {
```

```
  alert("Module loaded (export default)!");
}
```

…То для доступа к нему нам следует взять свойство `default` объекта модуля:

```
let obj = await import('./say.js');
let say = obj.default;
// или, одной строкой: let {default: say} = await import('./
say.js');

say();
```

Вот полный пример:

https://plnkr.co/edit/FW9PulnQJs2WvRS4?p=preview

На заметку:

Динамический импорт работает в обычных скриптах, он не требует указания `script type="module"`.

На заметку:

Хотя `import()` и выглядит похоже на вызов функции, на самом деле это специальный синтаксис, так же, как, например, `super()`.

Так что мы не можем скопировать `import` в другую переменную или вызвать при помощи `.call/apply`. Это не функция.

Разное

Proxy и Reflect

Объект `Proxy` «оборачивается» вокруг другого объекта и может перехватывать (и, при желании, самостоятельно обрабатывать) разные действия с ним, например чтение/запись свойств и другие. Далее мы будем называть такие объекты «прокси».

Прокси используются во многих библиотеках и некоторых браузерных фреймворках. В этой главе мы увидим много случаев применения прокси в решении реальных задач.

Синтаксис:

```
let proxy = new Proxy(target, handler);
```

> `target` – это объект, для которого нужно сделать прокси, может быть чем угодно, включая функции.
>
> `handler` – конфигурация прокси: объект с «ловушками» («traps»): методами, которые перехватывают разные операции, например, ловушка `get` – для чтения свойства из `target`, ловушка `set` – для записи свойства в `target` и так далее.

При операциях над `proxy`, если в `handler` имеется соответствующая «ловушка», то она срабатывает, и прокси имеет возможность по-своему обработать её, иначе операция будет совершена над оригинальным объектом `target`.

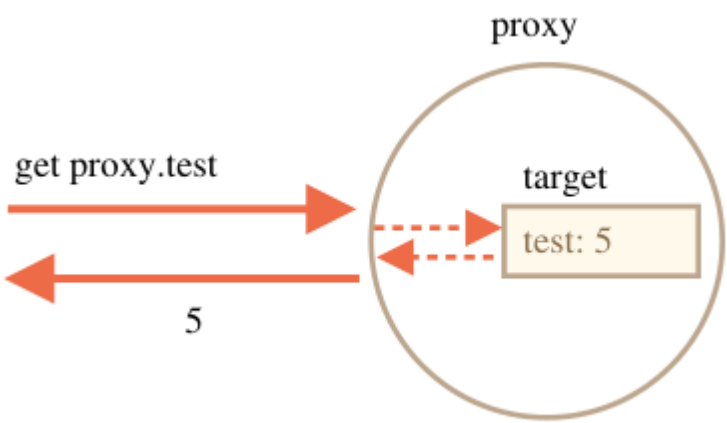
В качестве начального примера создадим прокси без всяких ловушек:

```
let target = {};
let proxy = new Proxy(target, {}); // пустой handler

proxy.test = 5; // записываем в прокси (1)
alert(target.test); // 5, свойство появилось в target!

alert(proxy.test); // 5, мы также можем прочитать его из
прокси (2)

for(let key in proxy) alert(key); // test, итерация работает
(3)
```

Так как нет ловушек, то все операции на `proxy` применяются к оригинальному объекту `target`.

1. Запись свойства `proxy.test=` устанавливает значение на `target`.
2. Чтение свойства `proxy.test` возвращает значение из `target`.
3. Итерация по `proxy` возвращает значения из `target`.

Как мы видим, без ловушек `proxy` является прозрачной обёрткой над `target`.



`Proxy` – это особый, «экзотический», объект, у него нет собственных свойств. С пустым `handler` он просто перенаправляет все операции на `target`.

Чтобы активировать другие его возможности, добавим ловушки.

Что именно мы можем ими перехватить?

Для большинства действий с объектами в спецификации JavaScript есть так называемый «внутренний метод», который на самом низком уровне описывает, как его выполнять. Например, `[[Get]]` – внутренний метод для чтения свойства, `[[Set]]` – для записи свойства, и так далее. Эти методы используются только в спецификации, мы не можем обратиться напрямую к ним по имени.

Ловушки как раз перехватывают вызовы этих внутренних методов. Полный список методов, которые можно перехватывать, перечислен в спецификации Proxy, а также в таблице ниже.

Для каждого внутреннего метода в этой таблице указана ловушка, то есть имя метода, который мы можем добавить в параметр `handler` при создании `new Proxy`, чтобы перехватывать данную операцию:

Внутренний метод	Ловушка	Что вызывает
`[[Get]]`	`get`	чтение свойства
`[[Set]]`	`set`	запись свойства
`[[HasProperty]]`	`has`	оператор `in`
`[[Delete]]`	`deleteProperty`	оператор `delete`
`[[Call]]`	`apply`	вызов функции
`[[Construct]]`	`construct`	оператор `new`
`[[GetPrototypeOf]]`	`getPrototypeOf`	Object.getPrototypeOf
`[[SetPrototypeOf]]`	`setPrototypeOf`	Object.setPrototypeOf
`[[IsExtensible]]`	`isExtensible`	Object.isExtensible
`[[PreventExtensions]]`	`preventExtensions`	Object.preventExtensio
`[[DefineOwnProperty]]`	`defineProperty`	Object.defineProperty,

		Object.defineProperties
`[[GetOwnProperty]]`	`getOwnPropertyDescriptor`	Object.getOwnProperty for..in, Object.keys/ values/ entries
`[[OwnPropertyKeys]]`	`ownKeys`	Object.getOwnProperty Object.getOwnProperty for..in, Object.keys/ values/ entries

Инварианты

JavaScript налагает некоторые условия – инварианты на реализацию внутренних методов и ловушек.

Большинство из них касаются возвращаемых значений:

- Метод `[[Set]]` должен возвращать `true`, если значение было успешно записано, иначе `false`.
- Метод `[[Delete]]` должен возвращать `true`, если значение было успешно удалено, иначе `false`.
- …и так далее, мы увидим больше в примерах ниже.

Есть и другие инварианты, например:

- Метод `[[GetPrototypeOf]]`, применённый к прокси, должен возвращать то же значение,

что и метод `[[GetPrototypeOf]]`, применённый к оригинальному объекту. Другими словами, чтение прототипа объекта прокси всегда должно возвращать прототип оригинального объекта.

Ловушки могут перехватывать вызовы этих методов, но должны выполнять указанные условия.

Инварианты гарантируют корректное и последовательное поведение конструкций и методов языка. Полный список инвариантов можно найти в спецификации, хотя скорее всего вы не нарушите эти условия, если только не соберётесь делать что-то совсем уж странное.

Теперь давайте посмотрим, как это всё работает, на реальных примерах.

Значение по умолчанию с ловушкой «get»

Чаще всего используются ловушки на чтение/запись свойств.

Чтобы перехватить операцию чтения, `handler` должен иметь метод `get(target, property, receiver)`.

Он срабатывает при попытке прочитать свойство объекта, с аргументами:

`target` – это оригинальный объект, который передавался первым аргументом в конструктор `new Proxy`,

`property` – имя свойства,

`receiver` – если свойство объекта является геттером, то `receiver` – это объект, который будет использован

как `this` при его вызове. Обычно это сам объект прокси (или наследующий от него объект). Прямо сейчас нам не понадобится этот аргумент, подробнее разберём его позже.

Давайте применим ловушку `get`, чтобы реализовать «значения по умолчанию» для свойств объекта.

Например, сделаем числовой массив, так чтобы при чтении из него несуществующего элемента возвращался `0`.

Обычно при чтении из массива несуществующего свойства возвращается `undefined`, но мы обернём обычный массив в прокси, который перехватывает операцию чтения свойства из массива и возвращает `0`, если такого элемента нет:

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
  get(target, prop) {
    if (prop in target) {
      return target[prop];
    } else {
      return 0; // значение по умолчанию
    }
  }
});

alert( numbers[1] ); // 1
alert( numbers[123] ); // 0 (нет такого элемента)
```

Как видно, это очень легко сделать при помощи ловушки `get`.

Мы можем использовать `Proxy` для реализации любой логики возврата значений по умолчанию.

Представим, что у нас есть объект-словарь с фразами на английском и их переводом на испанский:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome'] ); // undefined
```

Сейчас, если фразы в `dictionary` нет, при чтении возвращается `undefined`. Но на практике оставлять фразы непереведёнными лучше, чем использовать `undefined`. Поэтому давайте сделаем так, чтобы при отсутствии перевода возвращалась оригинальная фраза на английском вместо `undefined`.

Чтобы достичь этого, обернём `dictionary` в прокси, перехватывающий операцию чтения:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

dictionary = new Proxy(dictionary, {

    if (phrase in target) { // если перевод для фразы есть в словаре
      return target[phrase]; // возвращаем его
    } else {
      // иначе возвращаем непереведённую фразу
      return phrase;
    }
  }
});

// Запросим перевод произвольного выражения в словаре!
// В худшем случае оно не будет переведено
alert( dictionary['Hello'] ); // Hola
  get(target, phrase) { // перехватываем чтение свойства в dictionary
alert( dictionary['Welcome to Proxy']); // Welcome to Proxy (нет перевода)
```

Прокси следует использовать везде вместо `target`

Пожалуйста, обратите внимание: прокси перезаписывает переменную:

```
dictionary = new Proxy(dictionary, ...);
```

Прокси должен заменить собой оригинальный объект повсюду. Никто не должен ссылаться на оригинальный объект после того, как он был проксирован. Иначе очень легко запутаться.

Валидация с ловушкой «set»

Допустим, мы хотим сделать массив исключительно для чисел. Если в него добавляется значение иного типа, то это должно приводить к ошибке.

Ловушка `set` срабатывает, когда происходит запись свойства.

`set(target, property, value, receiver)`:

- `target` – это оригинальный объект, который передавался первым аргументом в конструктор `new Proxy`,
- `property` – имя свойства,
- `value` – значение свойства,
- `receiver` – аналогично ловушке `get`, этот аргумент имеет значение, только если свойство – сеттер.

Ловушка `set` должна вернуть `true`, если запись прошла успешно, и `false` в противном случае (будет сгенерирована ошибка `TypeError`).

Давайте применим её для проверки новых значений:

```
let numbers = [];

numbers = new Proxy(numbers, { // (*)

    if (typeof val == 'number') {
      target[prop] = val;
      return true;
    } else {
      return false;
    }
  }
});

numbers.push(1); // добавилось успешно
numbers.push(2); // добавилось успешно
alert("Длина: " + numbers.length); // 2



alert("Интерпретатор никогда не доходит до этой строки (из-за ошибки в строке выше)");
  set(target, prop, val) { // для перехвата записи свойства
numbers.push("тест"); // TypeError (ловушка set на прокси вернула false)
```

Обратите внимание, что встроенная функциональность массива по-прежнему работает! Значения добавляются методом `push`. Свойство `length` при этом увеличивается. Наш прокси ничего не ломает.

Нам не нужно переопределять методы массива `push` и `unshift` и другие, чтобы добавлять туда проверку на тип, так как внутри себя они используют операцию `[[Set]]`, которая перехватывается прокси.

Таким образом, код остаётся чистым и прозрачным.

Не забывайте вернуть `true`

Как сказано ранее, нужно соблюдать инварианты.

Для `set` реализация ловушки должна возвращать `true` в случае успешной записи свойства.

Если забыть это сделать или возвратить любое ложное значение, это приведёт к ошибке `TypeError`.

Перебор при помощи «ownKeys» и «getOwnPropertyDescriptor»

`Object.keys`, цикл `for..in` и большинство других методов, которые работают со списком свойств объекта, используют внутренний метод `[[OwnPropertyKeys]]` (перехватываемый ловушкой `ownKeys`) для их получения.

Такие методы различаются в деталях:

- `Object.getOwnPropertyNames(obj)` возвращает не-символьные ключи.
- `Object.getOwnPropertySymbols(obj)` возвращает символьные ключи.
- `Object.keys/values()` возвращает не-символьные ключи/значения с флагом `enumerable` (подробнее про флаги свойств было в главе Флаги и дескрипторы свойств).
- `for..in` перебирает не-символьные ключи с флагом `enumerable`, а также ключи прототипов.

…Но все они начинают с этого списка.

В примере ниже мы используем ловушку `ownKeys`, чтобы цикл `for..in` по объекту, равно как `Object.keys` и `Object.values` пропускали свойства, начинающиеся с подчёркивания `_`:

```
let user = {
  name: "Вася",
  age: 30,
  _password: "***"
};

user = new Proxy(user, {

    return Object.keys(target).filter(key => !
key.startsWith('_'));
  }
});

// ownKeys исключил _password
for(let key in user) alert(key); // name, затем: age

// аналогичный эффект для этих методов:
alert( Object.keys(user) ); // name,age
alert( Object.values(user) ); // Вася,30
  ownKeys(target) {
```

Как видно, работает.

Впрочем, если мы попробуем возвратить ключ, которого в объекте на самом деле нет, то `Object.keys` его не выдаст:

```
let user = { };

user = new Proxy(user, {

    return ['a', 'b', 'c'];
  }
});

  ownKeys(target) {
alert( Object.keys(user) ); // <пусто>
```

Почему? Причина проста: `Object.keys` возвращает только свойства с флагом `enumerable`. Для того, чтобы определить, есть ли этот флаг, он для каждого свойства вызывает внутренний метод `[[GetOwnProperty]]`, который получает его дескриптор. А в данном случае свойство отсутствует, его дескриптор пуст, флага `enumerable` нет, поэтому оно пропускается.

Чтобы `Object.keys` возвращал свойство, нужно либо чтобы свойство в объекте физически было, причём с флагом `enumerable`, либо перехватить вызовы `[[GetOwnProperty]]` (это делает ловушка `getOwnPropertyDescriptor`), и там вернуть дескриптор с `enumerable: true`.

Вот так будет работать:

```
let user = { };

user = new Proxy(user, {
  ownKeys(target) { // вызывается 1 раз для получения списка свойств
    return ['a', 'b', 'c'];
  },

  getOwnPropertyDescriptor(target, prop) { // вызывается для каждого свойства
    return {
      enumerable: true,
      configurable: true
      /* ...другие флаги, возможно, "value: ..." */
    };
  }

});

alert( Object.keys(user) ); // a, b, c
```

Ещё раз заметим, что получение дескриптора нужно перехватывать только если свойство отсутствует в самом объекте.

Защищённые свойства с ловушкой «deleteProperty» и другими

Существует широко распространённое соглашение о том, что свойства и методы, название которых начинается с символа подчёркивания `_`, следует считать внутренними. К ним не следует обращаться снаружи объекта.

Однако технически это всё равно возможно:

```
let user = {
  name: "Вася",
  _password: "secret"
};

alert(user._password); // secret
```

Давайте применим прокси, чтобы защитить свойства, начинающиеся на `_`, от доступа извне.

Нам будут нужны следующие ловушки:

- `get` – для того, чтобы сгенерировать ошибку при чтении такого свойства,
- `set` – для того, чтобы сгенерировать ошибку при записи,
- `deleteProperty` – для того, чтобы сгенерировать ошибку при удалении,
- `ownKeys` – для того, чтобы исключить такие свойства из `for..in` и методов типа `Object.keys`.

Вот соответствующий код:

```
let user = {
  name: "Вася",
  _password: "***"
};

user = new Proxy(user, {

    if (prop.startsWith('_')) {
      throw new Error("Отказано в доступе");
    } else {
      let value = target[prop];
      return (typeof value === 'function') ?
value.bind(target) : value; // (*)
    }
  },

    if (prop.startsWith('_')) {
      throw new Error("Отказано в доступе");
```

```
    } else {
      target[prop] = val;
      return true;
    }
  },

    if (prop.startsWith('_')) {
      throw new Error("Отказано в доступе");
    } else {
      delete target[prop];
      return true;
    }
  },

    return Object.keys(target).filter(key => !
key.startsWith('_'));
  }
});

// "get" не позволяет прочитать _password
try {
  alert(user._password); // Error: Отказано в доступе
} catch(e) { alert(e.message); }

// "set" не позволяет записать _password
try {
  user._password = "test"; // Error: Отказано в доступе
} catch(e) { alert(e.message); }

// "deleteProperty" не позволяет удалить _password
try {
  delete user._password; // Error: Отказано в доступе
} catch(e) { alert(e.message); }

// "ownKeys" исключает _password из списка видимых для
итерации свойств
for(let key in user) alert(key); // name
  get(target, prop) {
  set(target, prop, val) { // перехватываем
запись свойства
  deleteProperty(target, prop) { //
перехватываем удаление свойства
  ownKeys(target) { // перехватываем попытку
итерации
```

Обратите внимание на важную деталь в ловушке `get` на строке `(*)`:

```
get(target, prop) {
  // ...
  let value = target[prop];

}
  return (typeof value === 'function') ?
value.bind(target) : value; // (*)
```

Зачем для функции вызывать `value.bind(target)`?

Всё дело в том, что метод самого объекта, например `user.checkPassword()`, должен иметь доступ к свойству `_password`:

```
user = {
  // ...
  checkPassword(value) {
    // метод объекта должен иметь доступ на чтение _password
    return value === this._password;
  }
}
```

Вызов `user.checkPassword()` получает проксированный объект `user` в качестве `this` (объект перед точкой становится `this`), так что когда такой вызов обращается к `this._password`, ловушка `get` вступает в действие (она срабатывает при любом чтении свойства), и выбрасывается ошибка.

Поэтому мы привязываем контекст к методам объекта – оригинальный объект `target` в строке `(*)`. Тогда их дальнейшие вызовы будут использовать `target` в качестве `this`, без всяких ловушек.

Такое решение обычно работает, но не является идеальным, поскольку метод может передать оригинальный объект куда-то ещё, и возможна путаница: где изначальный объект, а где – проксированный.

К тому же, объект может проксироваться несколько раз (для добавления различных возможностей), и если передавать методу исходный, то могут быть

неожиданности.

Так что везде использовать такой прокси не стоит.

Приватные свойства в классах

Современные интерпретаторы JavaScript поддерживают приватные свойства в классах. Названия таких свойств должны начинаться с символа `#`. Они подробно описаны в главе Приватные и защищённые методы и свойства. Для них не нужны подобные прокси.

Впрочем, приватные свойства имеют свои недостатки. В частности, они не наследуются.

«В диапазоне» с ловушкой «has»

Давайте посмотрим ещё примеры.

Предположим, у нас есть объект `range`, описывающий диапазон:

```
let range = {
  start: 1,
  end: 10
};
```

Мы бы хотели использовать оператор `in`, чтобы проверить, что некоторое число находится в указанном диапазоне.

Ловушка `has` перехватывает вызовы `in`.

```
has(target, property)
```

`target` – это оригинальный объект, который

передавался первым аргументом в конструктор `new Proxy`,

`property` – имя свойства

Вот демо:

```
let range = {
  start: 1,
  end: 10
};

range = new Proxy(range, {

    return prop >= target.start && prop <= target.end
  }
});
```

```
  has(target, prop) {
alert(5 in range); // true
alert(50 in range); // false
```

Отлично выглядит, не правда ли? И очень просто в реализации.

Оборачиваем функции: «apply»

Мы можем оборачивать в прокси и функции.

Ловушка `apply(target, thisArg, args)` активируется при вызове прокси как функции:

`target` – это оригинальный объект (как мы помним, функция – это объект в языке JavaScript),

`thisArg` – это контекст `this`.

`args` – список аргументов.

Например, давайте вспомним декоратор `delay(f, ms)`, созданный нами в главе Декораторы и переадресация

вызова, call/apply.

Тогда мы обошлись без создания прокси. Вызов `delay(f, ms)` возвращал функцию, которая передавала вызовы `f` после `ms` миллисекунд.

Вот предыдущая реализация, на основе функции:

```
function delay(f, ms) {
  // возвращает обёртку, которая вызывает функцию f через
таймаут
  return function() { // (*)
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Привет, ${user}!`);
}

// после обёртки вызовы sayHi будут
срабатывать с задержкой в 3 секунды
sayHi = delay(sayHi, 3000);

sayHi("Вася"); // Привет, Вася! (через 3
секунды)
```

Как мы уже видели, это в целом работает. Функция-обёртка в строке `(*)` вызывает нужную функцию с указанной задержкой.

Но наша функция-обёртка не перенаправляет операции чтения/записи свойства и другие. После обёртывания доступ к свойствам оригинальной функции, таким как `name`, `length`, и другим, будет потерян.

```
function delay(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
```

```
  alert(`Привет, ${user}!`);
}


sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 1 (в функции length - это число аргументов в её объявлении)
alert(sayHi.length); // 0 (в объявлении функции-обёртки ноль аргументов)
```

Прокси куда более мощные в этом смысле, поскольку они перенаправляют всё к оригинальному объекту.

Давайте используем прокси вместо функции-обёртки:

```
function delay(f, ms) {
  return new Proxy(f, {
    apply(target, thisArg, args) {
      setTimeout(() => target.apply(thisArg, args), ms);
    }
  });
}

function sayHi(user) {
  alert(`Привет, ${user}!`);
}

sayHi = delay(sayHi, 3000);


sayHi("Вася"); // Привет, Вася! (через 3 секунды)
alert(sayHi.length); // 1 (*) прокси перенаправляет чтение свойства length на исходную функцию
```

Результат такой же, но сейчас не только вызовы, но и другие операции на прокси перенаправляются к оригинальной функции. Таким образом, операция чтения свойства `sayHi.length` возвращает корректное значение в строке `(*)` после проксирования.

Мы получили лучшую обёртку.

Существуют и другие ловушки: полный список есть в начале этой главы. Использовать их можно по аналогии с вышеописанными.

Reflect

`Reflect` – встроенный объект, упрощающий создание прокси.

Ранее мы говорили о том, что внутренние методы, такие как `[[Get]]`, `[[Set]]` и другие, существуют только в спецификации, что к ним нельзя обратиться напрямую.

Объект `Reflect` делает это возможным. Его методы – минимальные обёртки вокруг внутренних методов.

Вот примеры операций и вызовы `Reflect`, которые делают то же самое:

Операция	Вызов `Reflect`	Внутренний метод
`obj[prop]`	`Reflect.get(obj, prop)`	`[[Get]]`
`obj[prop] = value`	`Reflect.set(obj, prop, value)`	`[[Set]]`
`delete obj[prop]`	`Reflect.deleteProperty(obj, prop)`	`[[Delete]]`
`new F(value)`	`Reflect.construct(F, value)`	`[[Construct]]`
...	...	...

Например:

```
let user = {};

Reflect.set(user, 'name', 'Вася');

alert(user.name); // Вася
```

В частности, `Reflect` позволяет вызвать операторы (`new`, `delete`...) как функции (`Reflect.construct`, `Reflect.deleteProperty`, ...). Это интересная возможность, но здесь нам важно другое.

Для каждого внутреннего метода, перехватываемого `Proxy`, есть соответствующий метод в `Reflect`, который имеет такое же имя и те же аргументы, что и у ловушки `Proxy`.

Поэтому мы можем использовать `Reflect`, чтобы перенаправить операцию на исходный объект.

В этом примере обе ловушки `get` и `set` прозрачно (как будто их нет) перенаправляют операции чтения и записи на объект, при этом выводя сообщение:

```
let user = {
  name: "Вася",
};

user = new Proxy(user, {
  get(target, prop, receiver) {
    alert(`GET ${prop}`);

  },
  set(target, prop, val, receiver) {
    alert(`SET ${prop}=${val}`);

  }
});

let name = user.name; // выводит "GET name"
user.name = "Петя"; // выводит "SET name=Петя"
    return Reflect.get(target, prop,
receiver); // (1)
    return Reflect.set(target, prop, val,
receiver); // (2)
```

Здесь:

1. `Reflect.get` читает свойство объекта.
2. `Reflect.set` записывает свойство и возвращает `true` при успехе, иначе `false`.

То есть, всё очень просто – если ловушка хочет перенаправить вызов на объект, то достаточно вызвать `Reflect.<метод>` с теми же аргументами.

В большинстве случаев мы можем сделать всё то же самое и без `Reflect`, например, чтение свойства `Reflect.get(target, prop, receiver)` можно заменить на `target[prop]`. Но некоторые нюансы легко упустить.

Прокси для геттера

Рассмотрим конкретный пример, демонстрирующий, чем лучше `Reflect.get`, и заодно разберёмся, зачем в `get/set` нужен третий аргумент `receiver`, мы его ранее не использовали.

Допустим, у нас есть объект `user` со свойством `_name` и геттером для него.

Сделаем вокруг `user` прокси:

```
let user = {
  _name: "Гость",
  get name() {
    return this._name;
  }
};
```

```
alert(userProxy.name); // Гость
let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop];
  }
```

```
});
```

Ловушка `get` здесь «прозрачная», она возвращает свойство исходного объекта и больше ничего не делает. Для нашего примера этого вполне достаточно.

Казалось бы, всё в порядке. Но давайте немного усложним пример.

Если мы унаследуем от проксированного `user` объект `admin`, то мы увидим, что он ведёт себя некорректно:

```
let user = {
  _name: "Гость",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop]; // (*) target = user
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Админ"
};

// Ожидается: Админ
alert(admin.name); // выводится Гость (?!?)
```

Обращение к свойству `admin.name` должно возвращать строку `"Админ"`, а выводит `"Гость"`!

В чём дело? Может быть, мы делаем что-то не так с наследованием?

Но если убрать прокси, то всё будет работать как ожидается.

На самом деле, проблема в прокси, в строке `(*)`.

1. При чтении `admin.name`, так как в объекте `admin` нет свойства `name`, оно ищется в прототипе.

2. Прототипом является прокси `userProxy`.

3. При чтении из прокси свойства `name` срабатывает ловушка `get` и возвращает его из исходного объекта как `target[prop]` в строке `(*)`.

   Вызов `target[prop]`, если `prop` – это геттер, запускает его код в контексте `this=target`. Поэтому результатом является `this._name` из исходного объекта `target`, то есть из `user`.

Именно для исправления таких ситуаций нужен `receiver`, третий аргумент ловушки `get`. В нём хранится ссылка на правильный контекст `this`, который нужно передать геттеру. В данном случае это `admin`.

Как передать геттеру контекст? Для обычной функции мы могли бы использовать `call/apply`, но это же геттер, его не вызывают, просто читают значение.

Это может сделать `Reflect.get`. Всё будет работать верно, если использовать его.

Вот исправленный вариант:

```
let user = {
  _name: "Гость",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) { // receiver = admin

  }
});
```

```
let admin = {
  __proto__: userProxy,
  _name: "Admин"
};
```

```
    return Reflect.get(target, prop,
receiver); // (*)
alert(admin.name); // Admин
```

Сейчас `receiver`, содержащий ссылку на корректный `this` (то есть на `admin`), передаётся геттеру посредством `Reflect.get` в строке `(*)`.

Можно переписать ловушку и короче:

```
get(target, prop, receiver) {
  return Reflect.get();
}
...
arguments
```

Методы в `Reflect` имеют те же названия, что и соответствующие ловушки, и принимают такие же аргументы. Это было специально задумано при разработке спецификации JavaScript.

Так что `return Reflect...` даёт простую и безопасную возможность перенаправить операцию на оригинальный объект и при этом предохраняет нас от возможных ошибок, связанных с этим действием.

Ограничения прокси

Прокси – уникальное средство для настройки поведения объектов на самом низком уровне. Но они не идеальны, есть некоторые ограничения.

Встроенные объекты: внутренние слоты

Многие встроенные объекты, например `Map`, `Set`, `Date`, `Promise` и другие используют так называемые «внутренние слоты».

Это как свойства, но только для внутреннего использования в самой спецификациии. Например, `Map` хранит элементы во внутреннем слоте `[[MapData]]`. Встроенные методы обращаются к слотам напрямую, не через `[[Get]]/[[Set]]`. Таким образом, прокси не может перехватить их.

Почему это имеет значение? Они же всё равно внутренние!

Есть один нюанс. Если встроенный объект проксируется, то в прокси не будет этих «внутренних слотов», так что попытка вызвать на таком прокси встроенный метод приведёт к ошибке.

Пример:

```
let map = new Map();

let proxy = new Proxy(map, {});

proxy.set('test', 1); // будет ошибка
```

Внутри себя объект типа `Map` хранит все данные во внутреннем слоте `[[MapData]]`. Прокси не имеет такого слота. Встроенный метод `Map.prototype.set` пытается получить доступ к своему внутреннему свойству `this.[[MapData]]`, но так как `this=proxy`, то не может его найти и завершается с ошибкой.

К счастью, есть способ исправить это:

```
let map = new Map();

let proxy = new Proxy(map, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
```

```
  }
});

proxy.set('test', 1);
alert(proxy.get('test')); // 1 (работает!)
    return typeof value == 'function' ?
value.bind(target) : value;
```

Сейчас всё сработало, потому что `get` привязывает свойства-функции, такие как `map.set`, к оригинальному объекту `map`. Таким образом, когда реализация метода `set` попытается получить доступ к внутреннему слоту `this.[[MapData]]`, то всё пройдёт благополучно.

Объект `Array` не использует внутренние слоты

Важным исключением является встроенный объект `Array`: он не использует внутренние слоты. Так сложилось исторически, ведь массивы были добавлены в язык очень давно.

То есть описанная выше проблема не возникает при проксировании массивов.

Приватные поля

Нечто похожее происходит и с приватными полями классов.

Например, метод `getName()` осуществляет доступ к приватному полю `#name`, после проксирования он перестаёт работать:

```
class User {
  #name = "Гость";

  getName() {
    return this.#name;
```

```
  }
}

let user = new User();

user = new Proxy(user, {});

alert(user.getName()); // Ошибка
```

Причина всё та же: приватные поля реализованы с использованием внутренних слотов. JavaScript не использует `[[Get]]/[[Set]]` при доступе к ним.

В вызове `getName()` значением `this` является проксированный `user`, в котором нет внутреннего слота с приватными полями.

Решением, как и в предыдущем случае, является привязка контекста к методу:

```
class User {
  #name = "Гость";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(target) :
value;
  }
});

alert(user.getName()); // Гость
```

Однако, такое решение имеет ряд недостатков, о которых уже говорилось: методу передаётся оригинальный объект, который может быть передан куда-то ещё, и это может поломать всю функциональность проксирования.

Прокси != оригинальный объект

Прокси и объект, который проксируется, являются двумя разными объектами. Это естественно, не правда ли?

Если мы используем оригинальный объект как ключ, а затем проксируем его, то прокси не будет найден:

```
let allUsers = new Set();

class User {
  constructor(name) {
    this.name = name;
    allUsers.add(this);
  }
}

let user = new User("Вася");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});

alert(allUsers.has(user)); // false
```

Как мы видим, после проксирования не получается найти объект `user` внутри множества `allUsers`, потому что прокси – это другой объект.

Прокси не перехватывают проверку на строгое равенство ===

Прокси способны перехватывать много операторов, например `new` (ловушка `construct`), `in` (ловушка `has`), `delete` (ловушка `deleteProperty`) и так далее.

Но нет способа перехватить проверку на строгое равенство. Объект строго равен только самому себе, и никаким другим значениям.

Так что все операции и встроенные классы,

которые используют строгую проверку объектов на равенство, отличат прокси от изначального объекта. Прозрачной замены в данном случае не произойдёт.

Отключаемые прокси

Отключаемый (revocable) прокси – это прокси, который может быть отключён вызовом специальной функции.

Допустим, у нас есть какой-то ресурс, и мы бы хотели иметь возможность закрыть к нему доступ в любой момент.

Для того, чтобы решить поставленную задачу, мы можем использовать отключаемый прокси, без ловушек. Такой прокси будет передавать все операции на проксируемый объект, и у нас будет возможность в любой момент отключить это.

Синтаксис:

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

Вызов возвращает объект с `proxy` и функцией `revoke`, которая отключает его.

Вот пример:

```
let object = {
  data: "Важные данные"
};

let {proxy, revoke} = Proxy.revocable(object, {});

// передаём прокси куда-нибудь вместо оригинального объекта...
alert(proxy.data); // Важные данные

// позже в коде
revoke();

// прокси больше не работает (отключён)
alert(proxy.data); // Ошибка
```

Вызов `revoke()` удаляет все внутренние ссылки на

оригинальный объект из прокси, так что между ними больше нет связи, и оригинальный объект теперь может быть очищен сборщиком мусора.

Мы можем хранить функцию `revoke` в `WeakMap`, чтобы легко найти её по объекту прокси:

```
let object = {
  data: "Важные данные"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ..позже в коде..
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Ошибка (прокси отключён)
let revokes = new WeakMap();
```

Преимущество такого подхода в том, что мы не должны таскать функцию `revoke` повсюду. Мы получаем её при необходимости из `revokes` по объекту прокси.

Мы использовали `WeakMap` вместо `Map`, чтобы не блокировать сборку мусора. Если прокси объект становится недостижимым (то есть на него больше нет ссылок), то `WeakMap` позволяет сборщику мусора удалить его из памяти вместе с соответствующей функцией `revoke`, которая в этом случае больше не нужна.

Ссылки

Спецификация: Proxy, Reflect.

MDN: Proxy, Reflect.

Итого

Прокси – это обёртка вокруг объекта, которая «по умолчанию» перенаправляет операции над ней на объект, но имеет возможность перехватывать их.

Проксировать можно любой объект, включая классы и функции.

Синтаксис:

```
let proxy = new Proxy(target, {
  /* ловушки */
});
```

…Затем обычно используют прокси везде вместо оригинального объекта `target`. Прокси не имеет собственных свойств или методов. Он просто перехватывает операцию, если имеется соответствующая ловушка, а иначе перенаправляет её сразу на объект `target`.

Мы можем перехватывать:

- Чтение (`get`), запись (`set`), удаление (`deleteProperty`) свойства (даже несуществующего).
- Вызов функции (`apply`).
- Оператор `new` (ловушка `construct`).
- И многие другие операции (полный список приведён в начале статьи, а также в документации).

Это позволяет нам создавать «виртуальные» свойства и методы, реализовывать значения по умолчанию, наблюдаемые объекты, функции-декораторы и многое другое.

Мы также можем оборачивать один и тот же объект много раз в разные прокси, добавляя ему различные аспекты

функциональности.

Reflect API создано как дополнение к Proxy. Для любой ловушки из `Proxy` существует метод в `Reflect` с теми же аргументами. Нам следует использовать его, если нужно перенаправить вызов на оригинальный объект.

Прокси имеют некоторые ограничения:

- Встроенные объекты используют так называемые «внутренние слоты», доступ к которым нельзя проксировать. Однако, ранее в этой главе был показан один способ, как обойти это ограничение.
- То же самое можно сказать и о приватных полях классов, так как они реализованы на основе слотов. То есть вызовы проксированных методов должны иметь оригинальный объект в качестве `this`, чтобы получить к ним доступ.
- Проверка объектов на строгое равенство `===` не может быть перехвачена.
- Производительность: конкретные показатели зависят от интерпретатора, но в целом получение свойства с помощью простейшего прокси занимает в несколько раз больше времени. В реальности это имеет значение только для некоторых «особо нагруженных» объектов.

Задачи

Ошибка при чтении несуществующего свойства

Обычно при чтении несуществующего свойства из объекта возвращается `undefined`.

Создайте прокси, который генерирует ошибку при попытке

прочитать несуществующее свойство.

Это может помочь обнаружить программные ошибки пораньше.

Напишите функцию `wrap(target)`, которая берёт объект `target` и возвращает прокси, добавляющий в него этот аспект функциональности.

Вот как это должно работать:

```
let user = {
  name: "John"
};

function wrap(target) {
  return new Proxy(target, {

  });
}

user = wrap(user);

alert(user.name); // John
        /* ваш код */
alert(user.age); // Ошибка: такого свойства не существует
```

К решению

Получение элемента массива с отрицательной позиции

В некоторых языках программирования возможно получать элементы массива, используя отрицательные индексы, отсчитываемые с конца.

Вот так:

```
let array = [1, 2, 3];

array[-1]; // 3, последний элемент
array[-2]; // 2, предпоследний элемент
array[-3]; // 1, за два элемента до последнего
```

Другими словами, `array[-N]` – это то же, что и `array[array.length - N]`.

Создайте прокси, который реализовывал бы такое поведение.

Вот как это должно работать:

```
let array = [1, 2, 3];

array = new Proxy(array, {
  /* ваш код */
});

alert( array[-1] ); // 3
alert( array[-2] ); // 2

// вся остальная функциональность массивов должна остаться без изменений
```

К решению

Observable

Создайте функцию `makeObservable(target)`, которая делает объект «наблюдаемым», возвращая прокси.

Вот как это должно работать:

```
function makeObservable(target) {
  /* ваш код */
}

let user = {};
user = makeObservable(user);
```

```
user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "John"; // выводит: SET name=John
```

Другими словами, возвращаемый `makeObservable` объект аналогичен исходному, но также имеет метод `observe(handler)`, который позволяет запускать `handler` при любом изменении свойств.

При изменении любого свойства вызывается `handler(key, value)` с именем и значением свойства.

P.S. В этой задаче ограничьтесь, пожалуйста, только записью свойства. Остальные операции могут быть реализованы похожим образом.

К решению

Eval: выполнение строки кода

Встроенная функция `eval` позволяет выполнять строку кода.

Синтаксис:

```
let result = eval(code);
```

Например:

```
let code = 'alert("Привет")';
```

```
eval(code); // Привет
```

Строка кода может быть большой, содержать переводы строк, объявления функций, переменные и т.п.

Результатом `eval` будет результат выполнения последней инструкции.

Например:

```
let value = eval('1+1');
alert(value); // 2
```

```
let value = eval('let i = 0; ++i');
alert(value); // 1
```

Код в `eval` выполняется в текущем лексическом окружении, поэтому ему доступны внешние переменные:

```
let a = 1;

function f() {
  let a = 2;


}

f();
  eval('alert(a)'); // 2
```

Значения внешних переменных можно изменять:

```
let x = 5;
eval("x = 10");
alert(x); // 10, значение изменено
```

В строгом режиме у `eval` имеется своё лексическое окружение. Поэтому функции и переменные, объявленные внутри `eval`, нельзя увидеть снаружи:

```
// напоминание: режим 'use strict' включён по умолчанию во всех исполняемых примерах

eval("let x = 5; function f() {}");

alert(typeof x); // undefined (нет такой переменной)
// функция f тоже невидима
```

Без `use strict` у `eval` не будет отдельного лексического окружения, поэтому `x` и `f` будут видны из внешнего кода.

Использование «eval»

В современной разработке на JavaScript `eval` используется весьма редко. Есть даже известное выражение – «eval is evil» («eval – это зло»).

Причина такого отношения достаточно проста: давным-давно JavaScript был не очень развитым языком, и многие вещи можно было сделать только с помощью `eval`. Но та эпоха закончилась более десяти лет назад.

На данный момент нет никаких причин, чтобы продолжать использовать `eval`. Если кто-то всё ещё делает это, то очень вероятно, что они легко смогут заменить `eval` более современными конструкциями или JavaScript-модулями.

Пожалуйста, имейте в виду, что код в `eval` способен получать доступ к внешним переменным, и это может иметь побочные эффекты.

Минификаторы кода (инструменты, используемые для сжатия JS-кода перед тем, как отправить его конечным пользователям) заменяют локальные переменные на другие с более короткими именами для оптимизации. Обычно это безопасная манипуляция, но не тогда, когда в коде используется `eval`, так как код из `eval` может изменять значения локальных переменных. Поэтому

минификаторы не трогают имена переменных, которые могут быть доступны из `eval`. Это ухудшает степень сжатия кода.

Использование внутри `eval` локальных переменных из внешнего кода считается плохим решением, так как это усложняет задачу по поддержке такого кода.

Существует два пути, как гарантированно избежать подобных проблем.

Если код внутри `eval` не использует внешние переменные, то вызывайте его так – `window.eval(...)`:

В этом случае код выполняется в глобальной области видимости:

```
let x = 1;
{
  let x = 5;
  window.eval('alert(x)'); // 1 (глобальная переменная)
}
```

Если коду внутри `eval` нужны локальные переменные, поменяйте `eval` на `new Function` и передавайте необходимые данные как аргументы:

```
let f = new Function('a', 'alert(a)');

f(5); // 5
```

Конструкция `new Function` объясняется в главе Синтаксис "new Function". Она создаёт функцию из строки в глобальной области видимости. Так что локальные переменные для неё невидимы, но всегда можно передать их как аргументы. Получается очень аккуратный код, как в примере выше.

Итого

Вызов `eval(code)` выполняет строку кода и возвращает результат последней инструкции.

- Это редко используется в современном JavaScript, так как в этом обычно нет необходимости.
- Возможен доступ к внешним локальным переменным. Это считается плохой практикой.
- Чтобы выполнить строку кода с помощью `eval` в глобальной области видимости, используйте `window.eval(code)`.
- Или же, если ваш код нуждается в каких-то данных из внешней области видимости, то используйте `new Function`, передав эти данные в качестве аргументов.

Задачи

Eval-калькулятор

важность: 4

Создайте калькулятор, который запрашивает ввод какого-нибудь арифметического выражения и возвращает результат его вычисления.

В этой задаче нет необходимости проверять полученное выражение на корректность, просто вычислить и вернуть результат.

Запустить демо

К решению

Каррирование

Каррирование – продвинутая техника для работы с функциями. Она используется не только в JavaScript, но и в других языках.

Каррирование – это трансформация функций таким образом, чтобы они принимали аргументы не как `f(a, b, c)`, а как `f(a)(b)(c)`.

Каррирование не вызывает функцию. Оно просто трансформирует её.

Давайте сначала посмотрим на пример, чтобы лучше понять, о чём речь, а потом на практическое применение каррирования.

Создадим вспомогательную функцию `curry(f)`, которая выполняет каррирование функции `f` с двумя аргументами. Другими словами, `curry(f)` для функции `f(a, b)` трансформирует её в `f(a)(b)`.

```
// использование
function sum(a, b) {
  return a + b;
}

let carriedSum = curry(sum);

alert( carriedSum(1)(2) ); // 3
function curry(f) { // curry(f) выполняет каррирование
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
}
```

Как вы видите, реализация довольна проста: это две обёртки.

> Результат `curry(func)` – обёртка `function(a)`.
>
> Когда она вызывается как `sum(1)`, аргумент сохраняется в лексическом окружении и возвращается новая обёртка `function(b)`.
>
> Далее уже эта обёртка вызывается с аргументом `2` и передаёт вызов к оригинальной функции `sum`.

Более продвинутые реализации каррирования, как например _.curry из библиотеки lodash, возвращают обёртку, которая позволяет запустить функцию как обычным образом, так и частично.

```
function sum(a, b) {
  return a + b;
}

let carriedSum = _.curry(sum); // используем _.curry из lodash

alert( carriedSum(1, 2) ); // 3, можно вызывать как обычно
alert( carriedSum(1)(2) ); // 3, а можно частично
```

Каррирование? Зачем?

Чтобы понять пользу от каррирования, нам определённо нужен пример из реальной жизни.

Например, у нас есть функция логирования `log(date, importance, message)`, которая форматирует и выводит информацию. В реальных проектах у таких функций есть много полезных возможностей, например, посылать логи по сети, здесь для простоты используем `alert`:

```
function log(date, importance, message) {
  alert(`[${date.getHours()}:${date.getMinutes()}]
[${importance}] ${message}`);
}
```

А теперь давайте применим к ней каррирование!

```
log = _.curry(log);
```

После этого `log` продолжает работать нормально:

```
log(new Date(), "DEBUG", "some debug"); // log(a, b, c)
```

…Но также работает вариант с каррированием:

```
log(new Date())("DEBUG")("some debug"); // log(a)(b)(c)
```

Давайте сделаем удобную функцию для логов с текущим временем:

```
// logNow будет частичным применением функции log с
фиксированным первым аргументом
let logNow = log(new Date());

// используем её
logNow("INFO", "message"); // [HH:mm] INFO message
```

Теперь `logNow` – это `log` с фиксированным первым аргументом, иначе говоря, «частично применённая» или «частичная» функция.

Мы можем пойти дальше и сделать удобную функцию для именно отладочных логов с текущим временем:

```
let debugNow = logNow("DEBUG");

debugNow("message"); // [HH:mm] DEBUG message
```

Итак:

1. Мы ничего не потеряли после каррирования: `log` всё так же можно вызывать нормально.
2. Мы можем легко создавать частично применённые

функции, как сделали для логов с текущим временем.

Продвинутая реализация каррирования

В случае, если вам интересны детали, вот «продвинутая» реализация каррирования для функций с множеством аргументов, которую мы могли бы использовать выше.

Она очень короткая:

```
function curry(func) {

  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      }
    }
  };

}
```

Примеры использования:

```
function sum(a, b, c) {
  return a + b + c;
}

let curriedSum = curry(sum);

alert( curriedSum(1, 2, 3) ); // 6, всё ещё можно вызывать
нормально
alert( curriedSum(1)(2,3) ); // 6, каррирование первого
аргумента
alert( curriedSum(1)(2)(3) ); // 6, каррирование всех
аргументов
```

Новое `curry` выглядит сложновато, но на самом деле его легко понять.

Результат вызова `curry(func)` – это обёртка `curried`, которая выглядит так:

```
// func -- функция, которую мы трансформируем
function curried(...args) {
  if (args.length >= func.length) { // (1)
    return func.apply(this, args);
  } else {
    return function pass(...args2) { // (2)
      return curried.apply(this, args.concat(args2));
    }
  }
};
```

Когда мы запускаем её, есть две ветви выполнения `if`:

1. Вызвать сейчас: если количество переданных аргументов `args` совпадает с количеством аргументов при объявлении функции (`func.length`) или больше, тогда вызов просто переходит к ней.
2. Частичное применение: в противном случае `func` не вызывается сразу. Вместо этого, возвращается другая обёртка `pass`, которая снова применит `curried`, передав предыдущие аргументы вместе с новыми. Затем при новом вызове мы опять получим либо новое частичное применение (если аргументов недостаточно) либо, наконец, результат.

Например, давайте посмотрим, что произойдёт в случае `sum(a, b, c)`. У неё три аргумента, так что `sum.length = 3`.

Для вызова `curried(1)(2)(3)`:

1. Первый вызов `curried(1)` запоминает `1` в своём лексическом окружении и возвращает обёртку `pass`.
2. Обёртка `pass` вызывается с `(2)`: она берёт предыдущие аргументы (`1`), объединяет их с тем, что получила сама `(2)` и вызывает `curried(1, 2)` со всеми ними. Так как число аргументов всё ещё меньше

3-х, `curry` возвращает `pass`.

3. Обёртка `pass` вызывается снова с `(3)`. Для следующего вызова `pass(3)` берёт предыдущие аргументы (`1`, `2`) и добавляет к ним `3`, делая вызов `curried(1, 2, 3)` – наконец 3 аргумента, и они передаются оригинальной функции.

Если всё ещё не понятно, просто распишите последовательность вызовов на бумаге.

Только функции с фиксированным количеством аргументов

Для каррирования необходима функция с фиксированным количеством аргументов.

Функцию, которая использует остаточные параметры, типа `f(...args)`, так каррировать не получится.

Немного больше, чем каррирование

По определению, каррирование должно превращать `sum(a, b, c)` в `sum(a)(b)(c)`.

Но, как было описано, большинство реализаций каррирования в JavaScript более продвинуты: они также оставляют вариант вызова функции с несколькими аргументами.

Итого

Каррирование – это трансформация, которая превращает

вызов `f(a, b, c)` в `f(a)(b)(c)`. В JavaScript реализация обычно позволяет вызывать функцию обоими вариантами: либо нормально, либо возвращает частично применённую функцию, если количество аргументов недостаточно.

Каррирование позволяет легко получать частичные функции. Как мы видели в примерах с логами: универсальная функция `log(date, importance, message)` после каррирования возвращает нам частично применённую функцию, когда вызывается с одним аргументом, как `log(date)` или двумя аргументами, как `log(date, importance)`.

Побитовые операторы

Побитовые операторы интерпретируют операнды как последовательность из 32 битов (нулей и единиц). Они производят операции, используя двоичное представление числа, и возвращают новую последовательность из 32 бит (число) в качестве результата.

Эта глава требует дополнительных знаний в программировании и не очень важная, при первом чтении вы можете пропустить её и вернуться потом, когда захотите понять, как побитовые операторы работают.

Формат 32-битного целого числа со знаком

Побитовые операторы в JavaScript работают с 32-битными целыми числами в их двоичном представлении.

Это представление называется «32-битное целое со знаком, старшим битом слева и дополнением до двойки».

Разберём, как устроены числа внутри подробнее, это необходимо знать для битовых операций с ними.

Что такое двоичная система счисления, вам, надеюсь, уже известно. При разборе побитовых операций мы будем обсуждать именно двоичное представление чисел, из 32 бит.

Старший бит слева – это научное название для самого обычного порядка записи цифр (от большего разряда к меньшему). При этом, если больший разряд отсутствует, то соответствующий бит равен нулю.

Примеры представления чисел в двоичной системе:

```
a = 0;   // 00000000000000000000000000000000
a = 1;   // 00000000000000000000000000000001
a = 2;   // 00000000000000000000000000000010
a = 3;   // 00000000000000000000000000000011
a = 255;// 00000000000000000000000011111111
```

Обратите внимание, каждое число состоит ровно из 32-битов.

Дополнение до двойки – это название способа поддержки отрицательных чисел.

Двоичный вид числа, обратного данному (например, `5` и `-5`) получается путём обращения всех битов с прибавлением 1.

То есть, нули заменяются на единицы, единицы – на нули и к числу прибавляется `1`. Получается внутреннее представление того же числа, но со знаком минус.

Например, вот число `314`:

```
00000000000000000000000100111010
```

Чтобы получить `-314`, первый шаг – обратить биты числа: заменить `0` на `1`, а `1` на `0`:

```
11111111111111111111111011000101
```

Второй шаг – к полученному двоичному числу прибавить единицу, обычным двоичным сложением: `11111111111111111111111011000101 + 1 = 11111111111111111111111011000110`.

Итак, мы получили:

```
-314 = 11111111111111111111111011000110
```

Принцип дополнения до двойки делит все двоичные представления на два множества: если крайний-левый бит равен `0` – число положительное, если `1` – число отрицательное. Поэтому этот бит называется *знаковым битом*.

Список операторов

В следующей таблице перечислены все побитовые операторы. Далее операторы разобраны более подробно.

Оператор	Использование	Описание
Побитовое И (AND)	`a & b`	Ставит 1 на бит результата, для которого соответствующие биты операндов равны 1.
Побитовое ИЛИ (OR)	`a | b`	Ставит 1 на бит результата, для которого хотя бы один из соответствующих

		битов операндов равен 1.
Побитовое исключающее ИЛИ (XOR)	`a ^ b`	Ставит 1 на бит результата, для которого только один из соответствующих битов операндов равен 1 (но не оба).
Побитовое НЕ (NOT)	`~a`	Заменяет каждый бит операнда на противоположный.
Левый сдвиг	`a << b`	Сдвигает двоичное представление `a` на `b` битов влево, добавляя справа нули.
Правый сдвиг, переносящий знак	`a >> b`	Сдвигает двоичное представление `a` на `b` битов вправо, отбрасывая сдвигаемые биты.
Правый сдвиг с заполнением нулями	`a >>> b`	Сдвигает двоичное представление `a` на `b` битов вправо, отбрасывая сдвигаемые биты

и добавляя нули слева.

Побитовые операторы работают следующим образом:

1. Операнды преобразуются в 32-битные целые числа, представленные последовательностью битов. Дробная часть, если она есть, отбрасывается.
2. Для бинарных операторов – каждый бит в первом операнде рассматривается вместе с соответствующим битом второго операнда: первый бит с первым, второй со вторым и т.п. Оператор применяется к каждой паре бит, давая соответствующий бит результата.
3. Получившаяся в результате последовательность бит интерпретируется как обычное число.

Посмотрим, как работают операторы, на примерах.

Вспомогательные функции parseInt, toString

Для удобной работы с примерами в этой статье, если вы захотите протестировать что-то в консоли, пригодятся две функции.

`parseInt("11000", 2)` – переводит строку с двоичной записью числа в число.

`n.toString(2)` – получает для числа `n` запись в 2-ной системе в виде строки.

Например:

```
var access = parseInt("11000", 2); // получаем число из строки
```

```
alert( access ); // 24, число с таким 2-ным
представлением

var access2 = access.toString(2); // обратно
двоичную строку из числа

alert( access2 ); // 11000
```

Без них перевод в двоичную систему и обратно был бы куда менее удобен. Более подробно они разбираются в главе Числа.

& (Побитовое И)

Выполняет операцию И над каждой парой бит.

Результат `a & b` равен единице только когда оба бита `a` и `b` равны единице.

Таблица истинности для `&`:

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
  = 00000000000000000000000000001110 (по осн. 2)
                  --------------------------------
14 & 9 (по осн. 10)
  = 00000000000000000000000000001000 (по осн. 2)
  = 8 (по осн. 10)
```

| (Побитовое ИЛИ)

Выполняет операцию ИЛИ над каждой парой бит.
Результат `a | b` равен 1, если хотя бы один бит из `a,b` равен 1.

Таблица истинности для `|`:

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
  = 00000000000000000000000000001110 (по осн. 2)
                  --------------------------------
14 | 9 (по осн. 10)
  = 00000000000000000000000000001111 (по осн. 2)
  = 15 (по осн. 10)
```

^ (Исключающее ИЛИ)

Выполняет операцию «Исключающее ИЛИ» над каждой парой бит.

`a` Исключающее ИЛИ `b` равно 1, если только `a=1` или только `b=1`, но не оба одновременно `a=b=1`.

Таблица истинности для исключающего ИЛИ:

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Как видно, оно даёт 1, если ЛИБО слева `1`, ЛИБО справа `1`, но не одновременно. Поэтому его и называют

«исключающее ИЛИ».

Пример:

```
9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
14 (по осн. 10)
  = 00000000000000000000000000001110 (по осн. 2)
                   --------------------------------
14 ^ 9 (по осн. 10)
  = 00000000000000000000000000000111 (по осн. 2)
  = 7 (по осн. 10)
```

Исключающее ИЛИ в шифровании

Исключающее или можно использовать для шифрования, так как эта операция полностью обратима. Двойное применение исключающего ИЛИ с тем же аргументом даёт исходное число.

Иначе говоря, верна формула: `a ^ b ^ b == a`.

Пусть Вася хочет передать Пете секретную информацию `data`. Эта информация заранее превращена в число, например строка интерпретируется как последовательность кодов символов.

Вася и Петя заранее договариваются о числовом ключе шифрования `key`.

Алгоритм:

> Вася берёт двоичное представление `data` и делает операцию `data ^ key`. При необходимости `data` бьётся на части, равные по длине `key`, чтобы можно было провести побитовое ИЛИ `^` для каждой части. В JavaScript оператор `^` работает с 32-

битными целыми числами, так что `data` нужно разбить на последовательность таких чисел.

Результат `data ^ key` отправляется Пете, это шифровка.

Например, пусть в `data` очередное число равно `9`, а ключ `key` равен `1220461917`.

```
Данные: 9 в двоичном виде
00000000000000000000000000001001

Ключ: 1220461917 в двоичном виде
01001000101111101100010101011101

Результат операции 9 ^ key:
01001000101111101100010101010100
Результат в 10-ной системе (шифровка):
1220461908
```

Петя, получив очередное число шифровки `1220461908`, применяет к нему такую же операцию `^ key`.

Результатом будет исходное число `data`.

В нашем случае:

```
Полученная шифровка в двоичной системе:
9 ^ key = 1220461908
01001000101111101100010101010100

Ключ: 1220461917 в двоичном виде:
01001000101111101100010101011101

Результат операции 1220461917 ^ key:
00000000000000000000000000001001
Результат в 10-ной системе (исходное сообщение):
9
```

Конечно, такое шифрование поддаётся частотному анализу и другим методам

дешифровки, поэтому современные алгоритмы используют операцию XOR `^` как одну из важных частей более сложной многоступенчатой схемы.

~ (Побитовое НЕ)

Производит операцию НЕ над каждым битом, заменяя его на обратный ему.

Таблица истинности для НЕ:

a	~a
0	1
1	0

Пример:

```
 9 (по осн. 10)
  = 00000000000000000000000000001001 (по осн. 2)
              --------------------------------
~9 (по осн. 10)
  = 11111111111111111111111111110110 (по осн. 2)
  = -10 (по осн. 10)
```

Из-за внутреннего представления отрицательных чисел получается так, что `~n == -(n+1)`.

Например:

```
alert( ~3 ); // -4
alert( ~-1 ); // 0
```

<< (Битовый сдвиг влево)

Операторы битового сдвига принимают два операнда. Первый – это число для сдвига, а второй – количество

битов, которые нужно сдвинуть в первом операнде.

Оператор `<<` сдвигает первый операнд на указанное число битов влево. Лишние биты отбрасываются, справа добавляются нулевые биты.

Например, `9 << 2` даст `36`:

```
9 (по осн.10)
  = 00000000000000000000000000001001 (по осн.2)
                  --------------------------------
9 << 2 (по осн.10)
  = 00000000000000000000000000100100 (по осн.2)
  = 36 (по осн.10)
```

Операция `<< 2` сдвинула и отбросила два левых нулевых бита и добавила справа два новых нулевых.

Левый сдвиг почти равен умножению на 2

Битовый сдвиг `<< N` обычно имеет тот же эффект, что и умножение на два `N` раз, например:

```
alert( 3 << 1 ); // 6, умножение на 2
alert( 3 << 2 ); // 12, умножение на 2 два раза
alert( 3 << 3 ); // 24, умножение на 2 три раза
```

Конечно, следует иметь в виду, что побитовые операторы работают только с 32-битными числами, поэтому верхний порог такого «умножения» ограничен:

```
alert(10000000000 * 2); // 20000000000, обычное
умножение
alert(10000000000 << 1); //
```

```
-1474836480, отброшен крайний-левый бит
```

>> (Правый битовый сдвиг, переносящий знак)
Этот оператор сдвигает биты вправо, отбрасывая лишние. При этом слева добавляется *копия* крайнего-левого бита.

Знак числа (представленный крайним-левым битом) при этом не меняется, так как новый крайний-левый бит имеет то же значение, что и исходном числе.

Поэтому он назван «переносящим знак».

Например, `9 >> 2` даст `2`:

```
9 (по осн.10)
  = 00000000000000000000000000001001 (по осн.2)
                    --------------------------------
9 >> 2 (по осн.10)
  = 00000000000000000000000000000010 (по осн.2)
  = 2 (по осн.10)
```

Операция `>> 2` сдвинула вправо и отбросила два правых бита `01` и добавила слева две копии первого бита `00`.

Аналогично, `-9 >> 2` даст `-3`:

```
-9 (по осн.10)
  = 11111111111111111111111111110111 (по осн.2)
                    --------------------------------
-9 >> 2 (по осн.10)
  = 11111111111111111111111111111101 (по осн.2) = -3 (по
осн.10)
```

Здесь операция `>> 2` сдвинула вправо и отбросила два правых бита `11` и добавила слева две копии первого бита `11.` , Знак числа сохранён, так как крайний-левый (знаковый) бит сохранил значение `1`.

Правый сдвиг почти равен целочисленному делению на 2

Битовый сдвиг `>> N` обычно имеет тот же результат, что и целочисленное деление на два `N` раз:

```
alert( 100 >> 1 ); // 50, деление на 2
alert( 100 >> 2 ); // 25, деление на 2 два раза
alert( 100 >> 3 ); // 12, деление на 2 три раза,
целая часть от результата
```

>>> (Правый сдвиг с заполнением нулями)

Этот оператор сдвигает биты первого операнда вправо. Лишние биты справа отбрасываются. Слева добавляются нулевые биты.

Знаковый бит становится равным 0, поэтому результат всегда положителен.

Для неотрицательных чисел правый сдвиг с заполнением нулями `>>>` и правый сдвиг с переносом знака `>>` дадут одинаковый результат, т.к. в обоих случаях слева добавятся нули.

Для отрицательных чисел – результат работы разный. Например, `-9 >>> 2` даст `1073741821`, в отличие от `-9 >> 2` (даёт `-3`):

```
-9 (по осн.10)
  = 11111111111111111111111111110111 (по осн.2)
                  --------------------------------
-9 >>> 2 (по осн.10)
  = 00111111111111111111111111111101 (по осн.2)
  = 1073741821 (по осн.10)
```

Применение побитовых операторов

Побитовые операторы используются редко, но всё же

используются.

Случаи применения побитовых операторов, которые мы здесь разберём, составляют большинство всех использований в JavaScript.

Осторожно, приоритеты!

В JavaScript побитовые операторы `^`, `&`, `|` выполняются после сравнений `==`.

Например, в сравнении `a == b^0` будет сначала выполнено сравнение `a == b`, а потом уже операция `^0`, как будто стоят скобки `(a == b)^0`.

Обычно это не то, чего мы хотим. Чтобы гарантировать желаемый порядок, нужно ставить скобки: `a == (b^0)`.

Маска

Для этого примера представим, что наш скрипт работает с пользователями.

У них могут быть различные роли в проекте:

```
Гость
Редактор
Админ
```

Каждой роли соответствует ряд доступов к статьям и функциональности сайта.

Например, `Гость` может лишь просматривать статьи сайта, а `Редактор` – ещё и редактировать их, и тому подобное.

Что-то в таком духе:

Пользователь	Просмотр статей	Изменение статей	Просмотр товаров	Изменение товаров	Управление правами
Гость	Да	Нет	Да	Нет	Нет
Редактор	Да	Да	Да	Да	Нет
Админ	Да	Да	Да	Да	Да

Если вместо «Да» поставить `1`, а вместо «Нет» – `0`, то каждый набор доступов описывается числом:

Пользователь	Просмотр статей	Изменение статей	Просмотр товаров	Изменение товаров	Управление правами	В 10-ной системе
Гость	1	0	1	0	0	= 20
Редактор	1	1	1	1	0	= 30
Админ	1	1	1	1	1	= 31

В последней колонке находится десятичное число, которое получится, если прочитать строку доступов в двоичном виде.

Например, доступ гостя `10100 = 20`.

Такая интерпретация доступов позволяет «упаковать» много информации в одно число. Это экономит память, а кроме этого – это удобно, поскольку в дополнение к экономии – по такому значению очень легко проверить, имеет ли посетитель заданную *комбинацию доступов*!

Для этого посмотрим, как в 2-ной системе представляется каждый доступ в отдельности.

Доступ, соответствующий только управлению правами: `00001 (=1)` (все нули кроме `1` на позиции,

соответствующей этому доступу).

Доступ, соответствующий только изменению товаров: `00010 (=2)`.

Доступ, соответствующий только просмотру товаров: `00100 (=4)`.

Доступ, соответствующий только изменению статей: `01000 (=8)`.

Доступ, соответствующий только просмотру статей: `10000 (=16)`.

Доступ одновременно на просмотр и изменение статей – это двоичное число с `1` на соответствующих позициях, то есть `access = 11000`.

Как правило, доступы задаются в виде констант:

```
var ACCESS_ADMIN = 1;           // 00001
var ACCESS_GOODS_EDIT = 2;    // 00010
var ACCESS_GOODS_VIEW = 4;      // 00100
var ACCESS_ARTICLE_EDIT = 8; // 01000
var ACCESS_ARTICLE_VIEW = 16;  // 10000
```

Из этих констант получить нужную комбинацию доступов можно при помощи операции `|`.

```
var guest = ACCESS_ARTICLE_VIEW | ACCESS_GOODS_VIEW; // 10100
var editor = guest | ACCESS_ARTICLE_EDIT | ACCESS_GOODS_EDIT;
// 11110
var admin = editor | ACCESS_ADMIN; // 11111
```

Теперь, чтобы понять, есть ли в доступе `editor` нужный доступ, например управление правами – достаточно применить к нему побитовый оператор И (`&`) с соответствующей константой.

Ненулевой результат будет означать, что доступ есть:

```
alert(editor & ACCESS_ADMIN); // 0, доступа нет
```

```
alert(editor & ACCESS_ARTICLE_EDIT); // 8, доступ есть
```

Такая проверка работает, потому что оператор И ставит `1` на те позиции результата, на которых в обоих операндах стоит `1`.

Можно проверить один из нескольких доступов.

Например, проверим, есть ли права на просмотр ИЛИ изменение товаров. Соответствующие права задаются битом `1` на втором и третьем месте с конца, что даёт число `00110` (=`6` в 10-ной системе).

```
var check = ACCESS_GOODS_VIEW | ACCESS_GOODS_EDIT; // 6, 00110

alert( admin & check ); // не 0, значит есть доступ к
просмотру ИЛИ изменению
```

Битовой маской называют как раз комбинацию двоичных значений (`check` в примере выше), которая используется для проверки и выборки единиц на нужных позициях.

Маски могут быть весьма удобны.

В частности, их используют в функциях, чтобы одним параметром передать несколько «флагов», т.е. однобитных значений.

Пример вызова функции с маской:

```
// найти пользователей с правами на изменение товаров или
администраторов
findUsers(ACCESS_GOODS_EDIT | ACCESS_ADMIN);
```

Это довольно-таки коротко и элегантно, но, вместе с тем, применение масок налагает определённые ограничения. В частности, побитовые операторы в JavaScript работают только с 32-битными числами, а значит, к примеру, 33 доступа уже в число не упакуешь. Да и работа с двоичной системой счисления – как ни крути, менее удобна, чем с

десятичной или с обычными логическими значениями `true/false`.

Поэтому основная сфера применения масок – это быстрые вычисления, экономия памяти, низкоуровневые операции, связанные с рисованием из JavaScript (3d-графика), интеграция с некоторыми функциями ОС (для серверного JavaScript), и другие ситуации, когда уже существуют функции, требующие битовую маску.

Округление

Так как битовые операции отбрасывают десятичную часть, то их можно использовать для округления. Достаточно взять любую операцию, которая не меняет значение числа.

Например, двойное НЕ (~):

```
alert( ~~12.345 ); // 12
```

Подойдёт и Исключающее ИЛИ (^) с нулём:

```
alert( 12.345 ^ 0 ); // 12
```

Последнее даже более удобно, поскольку отлично читается:

```
alert(12.3 * 14.5 ^ 0); // (=178) "12.3 умножить на 14.5 "
и округлить
```

У побитовых операторов достаточно низкий приоритет, он меньше чем у остальной арифметики:

```
alert( 1.1 + 1.2 ^ 0 ); // 2, сложение выполнится раньше
округления
```

Проверка на –1

Внутренний формат 32-битных чисел устроен так, что для смены знака нужно все биты заменить на противоположные («обратить») и прибавить `1`.

Обращение битов – это побитовое НЕ (~). То есть, при таком формате представления числа `-n = ~n + 1`. Или, если перенести единицу: `~n = -(n+1)`.

Как видно из последнего равенства, `~n == 0` только если `n == -1`. Поэтому можно легко проверить равенство `n == -1`:

```
var n = 5;

if (~n) { // сработает, т.к. ~n = -(5+1) = -6
  alert( "n не -1" ); // выведет!
}
```

```
var n = -1;

if (~n) { // не сработает, т.к. ~n = -(-1+1) = 0
  alert( "...ничего не выведет..." );
}
```

Проверка на `-1` пригождается, например, при поиске символа в строке. Вызов `str.indexOf("подстрока")` возвращает позицию подстроки в `str`, или `-1` если не нашёл.

```
var str = "Проверка";

if (~str.indexOf("верка")) { // Сочетание "if (~...indexOf)" читается как "если найдено"
  alert( 'найдено!' );
}
```

Умножение и деление на степени 2

Оператор `a << b`, сдвигая биты, по сути умножает `a` на `2` в степени `b`.

Например:

```
alert( 1 << 2 ); // 1*(2*2) = 4
alert( 1 << 3 ); // 1*(2*2*2) = 8
alert( 3 << 3 ); // 3*(2*2*2) = 24
```

При этом следует иметь в виду, что максимальный верхний порог такого умножения меньше, чем обычно, так как побитовый оператор манипулирует 32-битными целыми, в то время как обычные операторы работают с числами длиной 64 бита.

Оператор сдвига в другую сторону `a >> b`, производит обратную операцию – целочисленное деление `a` на 2^b.

```
alert( 8 >> 2 ); // 2 = 8/4, убрали 2 нуля в двоичном представлении
alert( 11 >> 2 ); // 2, целочисленное деление (менее значимые биты просто отброшены)
```

Итого

Бинарные побитовые операторы: `&` `|` `^` `<<` `>>` `>>>`.

Унарный побитовый оператор один: `~`.

Как правило, битовое представление числа используется для:

Округления числа: `(12.34^0) = 12`.

Проверки на равенство `-1`: `if (~n) { n не -1 }`.

Упаковки нескольких битововых значений («флагов») в одно значение. Это экономит память и позволяет проверять наличие комбинации флагов одним оператором `&`.

Других ситуаций, когда нужны битовые маски.

Задачи

Побитовый оператор и значение

важность: 5

Почему побитовые операции в примерах ниже не меняют число? Что они делают внутри?

```
alert( 123 ^ 0 ); // 123
alert( 0 ^ 123 ); // 123
alert( ~~123 ); // 123
```

К решению

Проверка, целое ли число

важность: 3

Напишите функцию `isInteger(num)`, которая возвращает `true`, если `num` – целое число, иначе `false`.

Например:

```
alert( isInteger(1) ); // true
alert( isInteger(1.5) ); // false
alert( isInteger(-0.5) ); // false
```

К решению

Симметричны ли операции ^, |, &?

важность: 5

Верно ли, что для любых `a` и `b` выполняются равенства ниже?

```
(a ^ b) == (b ^ a)
```

```
(a & b) == (b & a)
(a | b) == (b | a)
```

Иными словами, при перемене мест – всегда ли результат останется тем же?

К решению

Почему результат разный?

важность: 5

Почему результат второго `alert'a` такой странный?

```
alert( 123456789 ^ 0 ); // 123456789
alert( 12345678912345 ^ 0 ); // 1942903641
```

К решению

BigInt

> **Новая возможность**
>
> Эта возможность была добавлена в язык недавно. Узнать, где есть поддержка, можно на https://caniuse.com/#feat=bigint.

`BigInt` – это специальный числовой тип, который предоставляет возможность работать с целыми числами произвольной длины.

Чтобы создать значение типа `BigInt`, необходимо добавить `n` в конец числового литерала или вызвать функцию `BigInt`, которая создаст число типа `BigInt` из переданного аргумента. Аргументом может быть число, строка и др.

```
const bigint = 1234567890123456789012345678901234567890n;

const sameBigint =
BigInt("1234567890123456789012345678901234567890");

const bigintFromNumber = BigInt(10); // то же самое, что и 10n
```

Математические операторы

`BigInt` можно использовать как обычные числа, к примеру:

```
alert(1n + 2n); // 3

alert(5n / 2n); // 2
```

Обратите внимание: операция деления `5/2` возвращает округлённый результат, без дробной части. Все операции с числами типа `bigint` возвращают `bigint`.

В математических операциях мы не можем смешивать `bigint` и обычные числа:

```
alert(1n + 2); // Error: Cannot mix BigInt and other types
```

Мы должны явно их конвертировать: используя либо `BigInt()`, либо `Number()`, например:

```
let bigint = 1n;
let number = 2;

// конвертируем number в bigint
alert(bigint + BigInt(number)); // 3
```

```
// конвертируем `bigint` в number
alert(Number(bigint) + number); // 3
```

Конвертирование `bigint` в число всегда происходит неявно и без генерации ошибок, но если значение `bigint` слишком велико и не подходит под тип `number`, то дополнительные биты будут отброшены, так что следует быть осторожными с такими преобразованиями.

К `BigInt` числам нельзя применить унарный оператор `+`

Унарный оператор `+value` является хорошо известным способом конвертировать произвольное значение `value` в число.

Данный оператор не поддерживается при работе с `BigInt` числами.

```
let bigint = 1n;

alert( +bigint ); // SyntaxError: Unexpected
identifier
```

Операции сравнения

Операции сравнения, такие как `<`, `>`, работают с `bigint` и обычными числами как обычно:

```
alert( 2n > 1n ); // true

alert( 2n > 1 ); // true
```

Пожалуйста, обратите внимание, что обычные и `bigint` числа принадлежат к разным типам, они могут быть равны

только при нестрогом сравнении `==`:

```
alert( 1 == 1n ); // true

alert( 1 === 1n ); // false
```

Логические операции

В `if` или любом другом логическом операторе `bigint` число ведёт себя как обычное число.

К примеру, в `if bigint 0n` преобразуется в `false`, другие значения преобразуются в `true`:

```
if (0n) {
  // никогда не выполнится
}
```

Логические операторы `||`, `&&` и другие также работают с `bigint` числами как с обычными числами:

```
alert( 1n || 2 ); // 1

alert( 0n || 2 ); // 2
```

Полифилы

Создание полифила для `BigInt` – достаточно непростая задача. Причина в том, что многие операторы в JavaScript, такие как `+`, `-` и др., ведут себя по-разному с `bigint` по сравнению с обычными числами.

К примеру, деление `bigint` числа всегда возвращает `bigint` (округлённое при необходимости).

Чтобы эмулировать такое поведение, полифил должен будет проанализировать код и заменить все такие

операторы на свои вызовы. Такая реализация будет тяжеловесной, не очень хорошей с точки зрения производительности.

Вот почему на данный момент нет хорошо реализованного полифила.

Существует обратное решение, предложеное разработчиками библиотеки JSBI.

Эта библиотека реализует большие числа, используя собственные методы. Мы можем использовать их вместо встроенных `BigInt`:

Операция	**Встроенный `BigInt`**	**JSBI**
Создание из `number`	`a = BigInt(789)`	`a = JSBI.BigInt(789)`
Сложение	`c = a + b`	`c = JSBI.add(a, b)`
Вычитание	`c = a - b`	`c = JSBI.subtract(a, b)`
…	…	…

…А затем использовать полифил (плагин Babel) для замены вызовов JSBI на встроенные `Bigint` для браузеров, которые их поддерживают.

Другими словами, данный подход предлагает использовать JSBI вместо встроенных `BigInt`. JSBI внутри себя работает с числами как с `BigInt`, эмулирует их с соблюдением всех требований спецификации. Таким образом, мы можем выполнять JSBI-код в интерпретаторах, которые не поддерживают `Bigint`, а для тех, которые поддерживают – полифил преобразует

вызовы в обычные `Bigint`.

Ссылки

MDN: BigInt.
Спецификация: BigInt.

Intl: интернационализация в JavaScript

Общая проблема строк, дат, чисел в JavaScript – они «не в курсе» языка и особенностей стран, где находится посетитель.

В частности:

Строки

При сравнении сравниваются коды символов, а это неправильно, к примеру, в русском языке оказывается, что `"ё" > "я"` и `"а" > "Я"`, хотя всем известно, что `я` – последняя буква алфавита и это она должна быть больше любой другой.

Даты

В разных странах принята разная запись дат. Где-то пишут 31.12.2014 (Россия), а где-то 12/31/2014 (США), где-то иначе.

Числа

В одних странах выводятся цифрами, в других – иероглифами, длинные числа разделяются где-то пробелом, где-то запятой.

Все современные браузеры, кроме IE10 (но есть

библиотеки и для него) поддерживают стандарт ECMA 402, предназначенный решить эти проблемы навсегда.

Основные объекты

`Intl.Collator`

Умеет правильно сравнивать и сортировать строки.

`Intl.DateTimeFormat`

Умеет форматировать дату и время в соответствии с нужным языком.

`Intl.NumberFormat`

Умеет форматировать числа в соответствии с нужным языком.

Локаль

Локаль – первый и самый важный аргумент всех методов, связанных с интернационализацией.

Локаль описывается строкой из трёх компонентов, которые разделяются дефисом:

1. Код языка.
2. Код способа записи.
3. Код страны.

На практике не всегда указаны три, обычно меньше:

1. `ru` – русский язык, без уточнений.
2. `en-GB` – английский язык, используемый в Англии (`GB`).

3. `en-US` – английский язык, используемый в США (`US`).
4. `zh-Hans-CN` – китайский язык (`zh`), записываемый упрощённой иероглифической письменностью (`Hans`), используемый в Китае.

Также через суффикс `-u-*` можно указать расширения локалей, например `"th-TH-u-nu-thai"` – тайский язык (`th`), используемый в Таиланде (`TH`), с записью чисел тайскими буквами (๐, ๑, ๒, ๓, ๔, ๕, ๖, ๗, ๘, ๙) .

Стандарт, который описывает локали – RFC 5464, языки описаны в IANA language registry.

Все методы принимают локаль в виде строки или массива, содержащего несколько локалей в порядке предпочтения.

Если локаль не указана или `undefined` – берётся локаль по умолчанию, установленная в окружении (браузере).

Подбор локали localeMatcher

`localeMatcher` – вспомогательная настройка, которую тоже можно везде указать, она определяет способ подбора локали, если желаемая недоступна.

У него два значения:

`"lookup"` – означает простейший порядок поиска путём обрезания суффикса, например `zh-Hans-CN` → `zh-Hans` → `zh` → локаль по умолчанию.

`"best fit"` – использует встроенные алгоритмы и предпочтения браузера (или другого окружения) для выбора подходящей локали.

По умолчанию стоит `"best fit"`.

Если локалей несколько, например `["zh-Hans-CN", "ru-RU"]` то `localeMatcher` пытается подобрать наиболее подходящую локаль для первой из списка (китайская), если не получается – переходит ко второй

(русской) и так далее. Если ни одной не нашёл, например на компьютере не совсем поддерживается ни китайский ни русский, то используется локаль по умолчанию.

Как правило, `"best fit"` является здесь наилучшим выбором.

Строки, Intl.Collator

Синтаксис:

```
// создание
let collator = new Intl.Collator([locales, [options]])
```

Параметры:

`locales`

Локаль, одна или массив в порядке предпочтения.

`options`

Объект с дополнительными настройками:

`localeMatcher` – алгоритм выбора подходящей локали.

`usage` – цель сравнения: сортировка `"sort"` или поиск `"search"`, по умолчанию `"sort"`.

`sensitivity` – чувствительность: какие различия в символах учитывать, а какие – нет, варианты:

`base` – учитывать только разные символы, без диакритических знаков и регистра, например: `a` ≠ `б`, `e` = `ё`, `a` = `A`.

`accent` – учитывать символы и диакритические знаки, например: `a` ≠ `б`, `e` ≠ `ё`, `a` = `A`.

`case` – учитывать символы и регистр, например: `a` ≠ `б`, `e` = `ё`, `a` ≠ `A`.

`variant` – учитывать всё: символ, диакритические знаки, регистр, например: `a` ≠ `б`, `e` ≠ `ё`, `a` ≠ `A`, используется по умолчанию.

`ignorePunctuation` – игнорировать знаки пунктуации: `true/false`, по умолчанию `false`.

`numeric` – использовать ли численное сравнение: `true/false`, если `true`, то будет `12 > 2`, иначе `12 < 2`.

`caseFirst` – в сортировке должны идти первыми прописные или строчные буквы, варианты: `"upper"` (прописные), `"lower"` (строчные) или `"false"` (стандартное для локали, также является значением по умолчанию). Не поддерживается IE11.

В подавляющем большинстве случаев подходят стандартные параметры, то есть `options` указывать не нужно.

Использование:

```
let result = collator.compare(str1, str2);
```

Результат `compare` имеет значение `1` (больше), `0` (равно) или `-1` (меньше).

Например:

```
let collator = new Intl.Collator();

alert( "ёжик" > "яблоко" ); // true (ёжик больше, что неверно)
alert( collator.compare("ёжик", "яблоко") ); // -1 (ёжик
меньше, верно)
```

Выше были использованы полностью стандартные настройки. Они различают регистр символа, но это

различие можно убрать, если настроить чувствительность `sensitivity`:

```
let collator = new Intl.Collator();
alert( collator.compare("ЁжиК", "ёжик") ); // 1, разные

let collator = new Intl.Collator(undefined, {
  sensitivity: "accent"
});
alert( collator.compare("ЁжиК", "ёжик") ); // 0, одинаковые
```

Даты, Intl.DateTimeFormat

Синтаксис:

```
// создание
let formatter = new Intl.DateTimeFormat([locales, [options]])
```

Первый аргумент – такой же, как и в `Collator`, а в объекте `options` мы можем определить, какие именно части даты показывать (часы, месяц, год…) и в каком формате.

Полный список свойств `options`:

Свойство	Описание	Возможные значения	По умолчанию
`localeMatcher`	Алгоритм подбора локали	`lookup`, `best fit`	`best fit`
`formatMatcher`	Алгоритм подбора формата	`basic`, `best fit`	`best fit`
`hour12`	Включать ли время в 12-часовом	`true` -- 12-часовой формат,	

	формате	`false` -- 24-часовой	
`timeZone`	Временная зона	Временная зона, например `Europe/Moscow`	`UTC`
`weekday`	День недели	`narrow`, `short`, `long`	
`era`	Эра	`narrow`, `short`, `long`	
`year`	Год	`2-digit`, `numeric`	`undefined` или `numeric`
`month`	Месяц	`2-digit`, `numeric`, `narrow`, `short`, `long`	`undefined` или `numeric`
`day`	День	`2-digit`, `numeric`	`undefined` или `numeric`
`hour`	Час	`2-digit`, `numeric`	
`minute`	Минуты	`2-digit`, `numeric`	
second	Секунды	`2-digit`, `numeric`	
`timeZoneName`	Название таймзоны (нет в IE11)	`short`, `long`	

Все локали обязаны поддерживать следующие наборы настроек:

weekday, year, month, day, hour, minute, second
weekday, year, month, day
year, month, day
year, month
month, day
hour, minute, second

Если указанный формат не поддерживается, то настройка `formatMatcher` задаёт алгоритм подбора наиболее близкого формата: `basic` – по стандартным правилам и `best fit` – по умолчанию, на усмотрение окружения (браузера).

Использование:

```
let dateString = formatter.format(date);
```

Например:

```
let date = new Date(2014, 11, 31, 12, 30, 0);

let formatter = new Intl.DateTimeFormat("ru");
alert( formatter.format(date) ); // 31.12.2014

let formatter = new Intl.DateTimeFormat("en-US");
alert( formatter.format(date) ); // 12/31/2014
```

Длинная дата, с настройками:

```
let date = new Date(2014, 11, 31, 12, 30, 0);

let formatter = new Intl.DateTimeFormat("ru", {
  weekday: "long",
  year: "numeric",
  month: "long",
  day: "numeric"
```

```
});

alert( formatter.format(date) ); // среда, 31 декабря 2014 г.
```

Только время:

```
let date = new Date(2014, 11, 31, 12, 30, 0);

let formatter = new Intl.DateTimeFormat("ru", {
  hour: "numeric",
  minute: "numeric",
  second: "numeric"
});

alert( formatter.format(date) ); // 12:30:00
```

Числа, Intl.NumberFormat

Форматтер `Intl.NumberFormat` умеет красиво форматировать не только числа, но и валюту, а также проценты.

Синтаксис:

```
let formatter = new Intl.NumberFormat([locales[, options]]);

formatter.format(number); // форматирование
```

Параметры, как и раньше – локаль и опции.

Список опций:

Свойство	Описание	Возможные значения	По умолчанию
`localeMatcher`	Алгоритм подбора локали	`lookup`, `best fit`	`best fit`
`style`	Стиль	`decimal`,	`decimal`

	форматирования	`percent`, `currency`	
`currency`	Алфавитный код валюты	См. Список кодов валюты, например `USD`	
`currencyDisplay`	Показывать валюту в виде кода, локализованного символа или локализованного названия	`code`, `symbol`, `name`	`symbol`
`useGrouping`	Разделять ли цифры на группы	`true`, `false`	`true`
`minimumIntegerDigits`	Минимальное количество цифр целой части	от `1` до `21`	`21`
`minimumFractionDigits`	Минимальное количество десятичных цифр	от `0` до `20`	для чисел и процентов `0`, для валюты зависит от кода.
`maximumFractionDigits`	Максимальное количество десятичных цифр	от `minimumFractionDigits` до `20`.	для чисел `max(minimumFra` `3)`, для процентов `0`, для валюты зависит от кода.

minimumSignificantDigits	Минимальное количество значимых цифр	от 1 до 21	1
maximumSignificantDigits	Максимальное количество значимых цифр	minimumSignificantDigits до 21	

Пример без опций:

```
let formatter = new Intl.NumberFormat("ru");
alert( formatter.format(1234567890.123) ); // 1 234 567 890,123
```

С ограничением значимых цифр (важны только первые 3):

```
let formatter = new Intl.NumberFormat("ru", {
  maximumSignificantDigits: 3
});
alert( formatter.format(1234567890.123) ); // 1 230 000 000
```

С опциями для валюты:

```
let formatter = new Intl.NumberFormat("ru", {
  style: "currency",
  currency: "GBP"
});

alert( formatter.format(1234.5) ); // 1 234,5 £
```

С двумя цифрами после запятой:

```
let formatter = new Intl.NumberFormat("ru", {
  style: "currency",
  currency: "GBP",
  minimumFractionDigits: 2
});
```

```
alert( formatter.format(1234.5) ); // 1 234,50 £
```

Методы в Date, String, Number

Методы форматирования также поддерживаются в обычных строках, датах, числах:

`String.prototype.localeCompare(that [, locales [, options]])`

Сравнивает строку с другой, с учётом локали, например:

```
let str = "ёжик";

alert( str.localeCompare("яблоко", "ru") ); // -1
```

`Date.prototype.toLocaleString([locales [, options]])`

Форматирует дату в соответствии с локалью, например:

```
let date = new Date(2014, 11, 31, 12, 00);

alert( date.toLocaleString("ru", { year: 'numeric', month: 'long' }) ); // Декабрь 2014
```

`Date.prototype.toLocaleDateString([locales [, options]])`

То же, что и выше, но опции по умолчанию включают в себя год, месяц, день

`Date.prototype.toLocaleTimeString([locales [, options]])`

То же, что и выше, но опции по умолчанию включают в себя часы, минуты, секунды

`Number.prototype.toLocaleString([locales [,`

options]])

Форматирует число, используя опции `Intl.NumberFormat`.

Все эти методы при запуске создают соответствующий объект `Intl.*` и передают ему опции, можно рассматривать их как укороченные варианты вызова.

Старые IE

В IE10 рекомендуется использовать полифил, например библиотеку https://github.com/andyearnshaw/Intl.js.

Задачи

Отсортируйте массив с буквой ё

важность: 5

Используя `Intl.Collator`, отсортируйте массив:

```
let animals = ["тигр", "ёж", "енот", "ехидна", "АИСТ", "ЯК"];

// ... ваш код ...

alert( animals ); // АИСТ,ёж,енот,ехидна,тигр,ЯК
```

В этом примере порядок сортировки не должен зависеть от регистра.

Что касается буквы `"ё"`, то мы следуем обычным правилам сортировки буквы ё, по которым «е» и «ё» считаются одной и той же буквой, за исключением случая, когда два слова отличаются только в позиции буквы «е» /

«ё» – тогда слово с «е» ставится первым.

К решению

Решения

Привет, мир!

Вызвать alert

Открыть решение в песочнице.

К условию

Покажите сообщение с помощью внешнего скрипта

HTML-код:

```
<!DOCTYPE html>
<html>

<body>

  <script src="alert.js"></script>

</body>

</html>
```

Для файла `alert.js` в той же папке:

```
alert("Я JavaScript!");
```

К условию

Работа с переменными

В коде ниже каждая строка решения соответствует одному элементу в списке задач.

```
let admin, name; // можно объявить две переменные через запятую

name = "Джон";

admin = name;

alert( admin ); // "Джон"
```

К условию

Придумайте правильные имена

Переменная для названия нашей планеты

Например:

```
let ourPlanetName = "Земля";
```

Обратите внимание, мы могли бы использовать короткое имя `planet`, но тогда будет непонятно, о какой планете мы говорим. Лучше описать содержимое переменной подробнее, по крайней мере, до тех пор, пока имя переменной неСтанетСлишкомДлинным.

Имя текущего посетителя:

```
let currentUserName = "Джон";
```

Опять же, мы могли бы укоротить название до `userName`, если мы точно знаем, что это текущий пользователь.

Современные редакторы и автодополнение ввода в них позволяют легко писать длинные названия переменных. Не экономьте буквы. Имена, состоящие из трёх слов, вполне нормальны.

Если в вашем редакторе нет автодополнения, воспользуйтесь другими.

К условию

Какие буквы (заглавные или строчные) использовать для имён констант?

Обычно мы используем буквы в верхнем регистре для констант, которые «жёстко закодированы». Или, другими словами, когда значение известно до выполнения скрипта и записывается непосредственно в код.

В нашем примере, `birthday` именно такая переменная. Поэтому мы можем использовать заглавные буквы.

В отличие от предыдущей, переменная `age` вычисляется во время выполнения скрипта. Сегодня у нас один возраст, а через год уже совсем другой. Она является константой, потому что не изменяется при выполнении кода. Но она является «менее константной», чем `birthday`: она вычисляется, поэтому мы должны сохранить её в нижнем регистре.

К условию

Типы данных

Шаблонные строки

Обратные кавычки позволяют вставить выражение внутри `${...}` в строку.

```
let name = "Ilya";

// выражение - число 1
alert( `hello ${1}` ); // hello 1

// выражение - строка "name"
alert( `hello ${"name"}` ); // hello name

// выражение - переменная, вставим её в строку
alert( `hello ${name}` ); // hello Ilya
```

К условию

Взаимодействие: alert, prompt, confirm

Простая страница

JavaScript-код:

```
let name = prompt("Ваше имя?", "");
alert(name);
```

Вся страница:

```
<!DOCTYPE html>
<html>
<body>

  <script>
    'use strict';

    let name = prompt("Ваше имя?", "");
    alert(name);
  </script>

</body>
</html>
```

К условию

Базовые операторы, математика

Постфиксная и префиксная формы

Ответ:

a = 2

b = 2

c = 2

d = 1

```
let a = 1, b = 1;

alert( ++a ); // 2, префиксная форма возвращает
новое значение
alert( b++ ); // 1, постфиксная форма возвращает
старое значение

alert( a ); // 2, значение увеличено один раз
alert( b ); // 2, значение увеличено один раз
```

К условию

Результат присваивания

Ответ:

a = 4 (умножено на 2)

x = 5 (вычислено как 1 + 4)

К условию

Преобразование типов

```
"" + 1 + 0 = "10" // (1)
"" - 1 + 0 = -1 // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
```

```
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
7 / 0 = Infinity
"  -9  " + 5 = "  -9  5" // (3)
"  -9  " - 5 = -14 // (4)
null + 1 = 1 // (5)
undefined + 1 = NaN // (6)
" \t \n" - 2 = -2 // (7)
```

1. Сложение со строкой `"" + 1` преобразует `1` к строке: `"" + 1 = "1"`, и в следующем случае `"1" + 0` работает то же самое правило.
2. Вычитание `-` (как и большинство математических операторов) работает только с числами, пустая строка `""` приводится к `0`.
3. Сложение со строкой превращает число `5` в строку и добавляет к строке.
4. Вычитание всегда преобразует к числу, значит строка `" -9 "` становится числом `-9` (пробелы по краям обрезаются).
5. `null` становится `0` после численного преобразования.
6. `undefined` становится `NaN` после численного преобразования.
7. Пробельные символы, такие как `\t` и `\n`, по краям строки игнорируются при преобразовании в число, так что строка `" \t \n"`, аналогично пустой строке, становится `0` после численного преобразования.

К условию

Исправьте сложение

Причина в том, что окно запроса возвращает пользовательский ввод как строку.

Поэтому переменные получают значения `"1"` и `"2"` соответственно.

```
let a = "1"; // prompt("Первое число?", 1);
let b = "2"; // prompt("Второе число?", 2);

alert(a + b); // 12
```

Нам нужно привести строки к числам перед применением оператора `+`. Например, с помощью `Number()` или вставки `+` перед ними.

Вставить `+` можно прямо перед `prompt`:

```
let a = +prompt("Первое число?", 1);
let b = +prompt("Второе число?", 2);

alert(a + b); // 3
```

Или внутри `alert`:

```
let a = prompt("Первое число?", 1);
let b = prompt("Второе число?", 2);

alert(+a + +b); // 3
```

В последнем варианте унарный и бинарный `+` используются вместе. Выглядит забавно, правда?

К условию

Операторы сравнения

```
5 > 4 → true
"ананас" > "яблоко" → false
"2" > "12" → true
undefined == null → true
undefined === null → false
null == "\n0\n" → false
null === +"\n0\n" → false
```

Разъяснения:

1. Очевидно, `true`.
2. Используется посимвольное сравнение, поэтому `false`. `"a"` меньше, чем `"я"`.
3. Снова посимвольное сравнение. Первый символ первой строки `"2"` больше, чем первый символ второй `"1"`.
4. Специальный случай. Значения `null` и `undefined` равны только друг другу при нестрогом сравнении.
5. Строгое сравнение разных типов, поэтому `false`.
6. Аналогично `(4)`, `null` равен только `undefined`.
7. Строгое сравнение разных типов.

К условию

if (строка с нулём)

Да, выведется.

Любая строка, кроме пустой (а строка `"0"` – не пустая), в логическом контексте становится `true`.

Можно запустить и проверить:

```
if ("0") {
  alert( 'Привет' );
}
```

К условию

Название JavaScript

```
<!DOCTYPE html>
<html>

<body>
  <script>
    'use strict';

    let value = prompt('Какое "официальное" название JavaScript?', '');

    if (value == 'ECMAScript') {
      alert('Верно!');
    } else {
      alert('Не знаете? ECMAScript!');
    }
  </script>
```

```
</body>

</html>
```

К условию

Покажите знак числа

```
let value = prompt('Введите число', 0);

if (value > 0) {
  alert( 1 );
} else if (value < 0) {
  alert( -1 );
} else {
  alert( 0 );
}
```

К условию

Перепишите 'if' в '?'

```
result = (a + b < 4) ? 'Мало' : 'Много';
```

К условию

Перепишите 'if..else' в '?'

```
let message = (login == 'Сотрудник') ? 'Привет' :
  (login == 'Директор') ? 'Здравствуйте' :
  (login == '') ? 'Нет логина' :
  '';
```

К условию

Что выведет alert (ИЛИ)?

Ответ: `2`, это первое значение, которое в логическом контексте даст `true`.

```
alert( null || 2 || undefined );
```

К условию

Что выведет alert (ИЛИ)?

Ответ: сначала `1`, затем `2`.

```
alert( alert(1) || 2 || alert(3) );
```

Вызов `alert` не возвращает значения, или, иначе говоря, возвращает `undefined`.

1. Первый оператор ИЛИ `||` выполнит первый `alert(1)`.
2. Получит `undefined` и пойдёт дальше, ко второму операнду в поисках истинного значения.
3. Так как второй операнд `2` является истинным, то вычисления завершатся, результатом `undefined || 2` будет `2`, которое будет выведено внешним `alert( .... )`.

Второй оператор `||` не будет выполнен, выполнение до `alert(3)` не дойдёт, поэтому `3` выведено не будет.

К условию

Что выведет alert (И)?

Ответ: `null`, потому что это первое «ложное» значение из списка.

```
alert( 1 && null && 2 );
```

К условию

Что выведет alert (И)?

Ответ: `1`, а затем `undefined`.

```
alert( alert(1) && alert(2) );
```

Вызов `alert` не возвращает значения, или, иначе говоря, возвращает `undefined`.

Поэтому до правого `alert` дело не дойдёт, вычисления закончатся на левом.

К условию

Что выведет этот код?

Ответ: `3`.

```
alert( null || 2 && 3 || 4 );
```

Приоритет оператора `&&` выше, чем `||`, поэтому он выполнится первым.

Результат `2 && 3 = 3`, поэтому выражение приобретает вид:

```
null || 3 || 4
```

Теперь результатом является первое истинное значение: `3`.

К условию

Проверка значения из диапазона

```
if (age >= 14 && age <= 90)
```

К условию

Проверка значения вне диапазона

Первый вариант:

```
if (!(age >= 14 && age <= 90))
```

Второй вариант:

```
if (age < 14 || age > 90)
```

К условию

Вопрос о "if"

Ответ: первое и третье выполнятся.

Подробности:

```
// Выполнится.
// Результат -1 || 0 = -1, в логическом контексте
true
if (-1 || 0) alert( 'first' );

// Не выполнится
// -1 && 0 = 0,  в логическом контексте false
if (-1 && 0) alert( 'second' );

// Выполнится
// оператор && имеет больший приоритет, чем ||
// так что -1 && 1 выполнится раньше
// вычисления: null || -1 && 1  ->  null || 1  ->
1
if (null || -1 && 1) alert( 'third' );
```

К условию

Проверка логина

```
let userName = prompt("Кто там?", '');

if (userName == 'Админ') {

  let pass = prompt('Пароль?', '');

  if (pass == 'Я главный') {
    alert( 'Здравствуйте!' );
  } else if (pass == '' || pass == null) {
    alert( 'Отменено' );
  } else {
    alert( 'Неверный пароль' );
  }

} else if (userName == '' || userName == null) {
```

```
  alert( 'Отменено' );
} else {
  alert( "Я вас не знаю" );
}
```

Обратите внимание на вертикальные отступы внутри блоков `if`. Они технически не требуются, но делают код более читаемым.

К условию

Циклы while и for

Последнее значение цикла

Ответ: `1`.

```
let i = 3;

while (i) {
  alert( i-- );
}
```

Каждое выполнение цикла уменьшает `i`. Проверка `while(i)` остановит цикл при `i = 0`.

Соответственно, будет такая последовательность шагов цикла («развернём» цикл):

```
let i = 3;

alert(i--); // выведет 3, затем уменьшит i до 2

alert(i--) // выведет 2, затем уменьшит i до 1

alert(i--) // выведет 1, затем уменьшит i до 0

// все, проверка while(i) не даст выполняться циклу
дальше
```

К условию

Какие значения выведет цикл while?

Задача демонстрирует, как постфиксные/ префиксные варианты могут повлиять на результат, когда используются в сравнениях.

1. **От 1 до 4**

```
let i = 0;
while (++i < 5) alert( i );
```

Первое значение: `i = 1`, так как операция `++i` сначала увеличит `i`, а потом уже произойдёт сравнение и выполнение `alert`.

Далее `2, 3, 4`… Значения выводятся одно за другим. Для каждого значения сначала происходит увеличение, а потом – сравнение, так как `++` стоит перед переменной.

При `i = 4` произойдёт увеличение `i` до `5`,

а потом сравнение `while (5 < 5)` – это неверно. Поэтому на этом цикл остановится, и значение `5` выведено не будет.

2. **От 1 до 5**

```
let i = 0;
while (i++ < 5) alert( i );
```

Первое значение: `i = 1`. Остановимся на нём подробнее. Оператор `i++` увеличивает `i`, возвращая старое значение, так что в сравнении `i++ < 5` будет участвовать старое `i = 0`.

Но последующий вызов `alert` уже не относится к этому выражению, так что получит новый `i = 1`.

Далее `2, 3, 4`… Для каждого значения сначала происходит сравнение, а потом – увеличение, и затем срабатывание `alert`.

Окончание цикла: при `i = 4` произойдёт сравнение `while (4 < 5)` – верно, после этого сработает `i++`, увеличив `i` до `5`, так что значение `5` будет выведено. Оно станет последним.

Значение `i = 5` последнее, потому что на следующем шаге `while (5 < 5)` ложно.

К условию

Какие значения выведет цикл for?

Ответ: от `0` до `4` в обоих случаях.

```
for (let i = 0; i < 5; ++i) alert( i );
```

```
for (let i = 0; i < 5; i++) alert( i );
```

Такой результат обусловлен алгоритмом работы `for`:

1. Выполнить единожды присваивание `i = 0` перед чем-либо (начало).
2. Проверить условие `i < 5`
3. Если `true` – выполнить тело цикла `alert(i)`, и затем `i++`

Увеличение `i++` выполняется отдельно от проверки условия `(2)`, значение `i` при этом не используется, поэтому нет никакой разницы между `i++` и `++i`.

К условию

Выведите чётные числа

```
for (let i = 2; i <= 10; i++) {
  if (i % 2 == 0) {
    alert( i );
  }
}
```

Для проверки на чётность мы здесь используем

оператор получения остатка от деления `%`.

К условию

Замените for на while

```
let i = 0;
while (i < 3) {
  alert( `number ${i}!` );
  i++;
}
```

К условию

Повторять цикл, пока ввод неверен

```
let num;

do {
  num = prompt("Введите число, большее 100?", 0);
} while (num <= 100 && num);
```

Цикл `do..while` повторяется, пока верны две проверки:

1. Проверка `num <= 100` – то есть, введённое число всё ещё меньше `100`.
2. Проверка `&& num` вычисляется в `false`, когда `num` имеет значение `null` или пустая строка `''`. В этом случае цикл `while` тоже нужно прекратить.

Кстати, сравнение `num <= 100` при вводе `null` даст `true`, так что вторая проверка

необходима.

К условию

Вывести простые числа

Существует множество алгоритмов решения этой задачи.

Давайте воспользуемся вложенными циклами:

```
Для всех i от 1 до 10 {
  проверить, делится ли число i на какое-либо из чисел до него
  если делится, то это i не подходит, берём следующее
  если не делится, то i - простое число
}
```

Решение с использованием метки:

```
let n = 10;

nextPrime:
for (let i = 2; i <= n; i++) { // Для всех i...

  for (let j = 2; j < i; j++) { // проверить, делится ли число..
    if (i % j == 0) continue nextPrime; // не подходит, берём следующее
  }

  alert( i ); // простое число
}
```

Конечно же, его можно оптимизировать с точки зрения производительности. Например, проверять все `j` не от `2` до `i`, а от `2` до квадратного корня из `i`. А для очень больших чисел – существуют более эффективные

специализированные алгоритмы проверки простоты числа, например квадратичное решето и решето числового поля.

К условию

Конструкция "switch"

Напишите "if", аналогичный "switch"

Если совсем точно следовать работе `switch`, то `if` должен выполнять строгое сравнение `'==='`.

Впрочем, для таких строк, подойдёт и обычное сравнение `'=='`.

```
if(browser == 'Edge') {
  alert("You've got the Edge!");
} else if (browser == 'Chrome'
  || browser == 'Firefox'
  || browser == 'Safari'
  || browser == 'Opera') {
  alert( 'Okay we support these browsers too' );
} else {
  alert( 'We hope that this page looks ok!' );
}
```

Обратите внимание: конструкция `browser == 'Chrome' || browser == 'Firefox' ...`

разбита на несколько строк для лучшей читаемости.

Но всё равно запись через `switch` нагляднее.

К условию

Переписать условия "if" на "switch"

Первые две проверки – обычный `case`, третья разделена на два `case`:

```
const number = +prompt('Введите число между 0 и 3',
'');

switch (number) {
  case 0:
    alert('Вы ввели число 0');
    break;

  case 1:
    alert('Вы ввели число 1');
    break;

  case 2:
  case 3:
    alert('Вы ввели число 2, а может и 3');

}
     break;
```

Обратите внимание: `break` внизу не обязателен, но ставится по «правилам хорошего тона».

Допустим, он не стоит. Есть шанс, что в будущем нам понадобится добавить в конец

ещё один `case`, например `case 4`, и мы, вполне вероятно, забудем этот `break` поставить. В результате выполнение `case 2`/`case 3` продолжится на `case 4` и будет ошибка.

К условию

Функции

Обязателен ли "else"?

Оба варианта функций работают одинаково, отличий нет.

К условию

Перепишите функцию, используя оператор '?' или '||'

Используя оператор `?`:

```
function checkAge(age) {
  return (age > 18) ? true : confirm('Родители
разрешили?');
}
```

Используя оператор `||` (самый короткий

вариант):

```
function checkAge(age) {
  return (age > 18) || confirm('Родители
разрешили?');
}
```

Обратите внимание, что круглые скобки вокруг `age > 18` не обязательны. Они здесь для лучшей читаемости кода.

К условию

Функция min(a, b)

Вариант решения с использованием `if`:

```
function min(a, b) {
  if (a < b) {
    return a;
  } else {
    return b;
  }
}
```

Вариант решения с оператором `?`:

```
function min(a, b) {
  return a < b ? a : b;
}
```

P.S. В случае равенства `a == b` не имеет значения, что возвращать.

К условию

Функция pow(x,n)

```
function pow(x, n) {
  let result = x;

  for (let i = 1; i < n; i++) {
    result *= x;
  }

  return result;
}

let x = prompt("x?", '');
let n = prompt("n?", '');

if (n < 1) {
  alert(`Степень ${n} не поддерживается,
используйте натуральное число`);
} else {
  alert( pow(x, n) );
}
```

К условию

Функции-стрелки, основы

Перепишите с использованием функции-стрелки

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "Вы согласны?",

);
   () => alert("Вы согласились."),
   () => alert("Вы отменили
```

```
выполнение.")
```

Выглядит короче и понятней, правда?

К условию

Плохой стиль

Вы могли заметить следующие недостатки, сверху вниз:

```
function pow(x,n)  // <- отсутствует пробел между аргументами
{  // <- фигурная скобка на отдельной строке
  let result=1;   // <- нет пробелов вокруг знака =
  for(let i=0;i<n;i++) {result*=x;}   // <- нет пробелов
  // содержимое скобок { ... } лучше вынести на отдельную строку
  return result;
}

let x=prompt("x?",''), n=prompt("n?",'') // <-- технически допустимо,
// но лучше написать в 2 строки, также нет пробелов и точки с запятой
if (n<0)  // <- нет пробелов, стоит добавить отступ в одну строку сверху
{   // <- фигурная скобка на отдельной строке
  // ниже - слишком длинная строка, лучше разбить для улучшения читаемости
  alert(`Степень ${n} не поддерживается, введите целую степень, большую 0`);
}
else // <- можно на одной строке, вместе: "} else {"
{
  alert(pow(x,n))  // вложенный вызов функции, нет пробелов и точки с запятой
}
```

Исправленный вариант:

```
function pow(x, n) {
  let result = 1;
```

```
  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}

let x = prompt("x?", "");
let n = prompt("n?", "");

if (n < 0) {
  alert(`Степень ${n} не поддерживается,
    введите целую степень, большую 0`);
} else {
  alert( pow(x, n) );
}
```

К условию

Автоматическое тестирование с использованием фреймворка Mocha

Что не так с этим тестом?

Тест демонстрирует один из соблазнов, с которым сталкиваются разработчики при их написании.

У нас тут, по сути, три теста, но они написаны как одна функция с тремя проверками.

Иногда так проще писать, но если произойдёт ошибка, то гораздо сложнее понять, что пошло

не так.

Если ошибка происходит посередине сложного потока выполнения, то нам придётся выяснять, какие данные были в этом месте. По сути, придётся *отлаживать тест*.

Гораздо лучше разбить тест на несколько блоков `it` и ясно описать входные и ожидаемые на выходе данные.

Примерно так:

```
describe("Возводит x в степень n", function() {
  it("5 в степени 1 будет 5", function() {
    assert.equal(pow(5, 1), 5);
  });

  it("5 в степени 2 будет 25", function() {
    assert.equal(pow(5, 2), 25);
  });

  it("5 в степени 3 будет 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});
```

Мы заменили один `it` на `describe` и группу блоков `it`. Теперь, если какой-либо из блоков завершится неудачно, мы точно увидим, с какими данными это произошло.

Также мы можем изолировать один тест и запускать только его, написав `it.only` вместо `it`:

```
describe("Возводит x в степень n", function() {
  it("5 в степени 1 будет 5", function() {
    assert.equal(pow(5, 1), 5);
```

```
  });


  it("5 в степени 3 будет 125", function() {
    assert.equal(pow(5, 3), 125);
  });
});
  // Mocha будет запускать только этот
блок
  it.only("5 в степени 2 будет 25",
function() {
    assert.equal(pow(5, 2), 25);
  });
```

К условию

Объекты

Привет, object

```
let user = {};
user.name = "John";
user.surname = "Smith";
user.name = "Pete";
delete user.name;
```

К условию

Проверка на пустоту

Просто в цикле перебираем свойства объекта и

возвращаем `false`, как только встречаем свойство.

```
function isEmpty(obj) {
  for (let key in obj) {
    // если тело цикла начнет выполняться - значит
в объекте есть свойства
    return false;
  }
  return true;
}
```

Открыть решение с тестами в песочнице.

К условию

Объекты-константы?

Конечно, это сработает без проблем.

Объявление `const` защищает только саму переменную от изменений.

Другими словами, `user` хранит ссылку на объект. И это не может быть изменено. Но содержимое объекта менять можно.

```
const user = {
  name: "John"
};


// Ошибка
user = 123;
// Работает!
user.name = "Pete";
```

К условию

Сумма свойств объекта

```
let salaries = {
  John: 100,
  Ann: 160,
  Pete: 130
};

let sum = 0;
for (let key in salaries) {
  sum += salaries[key];
}

alert(sum); // 390
```

К условию

Умножаем все числовые свойства на 2

```
function multiplyNumeric(obj) {
  for (let key in obj) {
    if (typeof obj[key] == 'number') {
      obj[key] *= 2;
    }
  }
}
```

Открыть решение с тестами в песочнице.

К условию

Методы объекта, "this"

Проверка синтаксиса

Ошибка!

Попробуйте запустить:

```
let user = {
  name: "Джон",
  go: function() { alert(this.name) }
}

(user.go)() // ошибка!
```

Сообщение об ошибке в большинстве браузеров не даёт понимания, что же пошло не так.

Ошибка появляется, потому что точка с запятой пропущена после `user = {...}`.

JavaScript не вставляет автоматически точку с запятой перед круглой скобкой `(user.go)()`, поэтому читает этот код так:

```
let user = { go:... }(user.go)()
```

Теперь мы тоже можем увидеть, что такое объединённое выражение синтаксически является вызовом объекта `{ go: ... }` как функции с аргументом `(user.go)`. И это происходит в той же строчке с объявлением переменной `let user`, т.е. объект `user` ещё даже не определён, поэтому получается ошибка.

Если мы вставим точку с запятой – всё

заработает:

```
let user = {
  name: "Джон",
  go: function() { alert(this.name) }
}

(user.go)() // Джон
;
```

Обратите внимание, что круглые скобки вокруг `(user.go)` ничего не значат. Обычно они определяют последовательность операций (оператор группировки), но здесь вызов метода через точку `.` срабатывает первым в любом случае, поэтому группировка ни на что не влияет. Только точка с запятой имеет значение.

К условию

Объясните значение "this"

Вот как это объясняется.

1. Это обычный вызов метода объекта через точку `.`, и `this` ссылается на объект перед точкой.

2. Здесь то же самое. Круглые скобки (оператор группировки) тут не изменяют порядок выполнения операций – доступ к методу через точку в любом случае срабатывает первым.

3. Здесь мы имеем более сложный вызов

`(expression).method()`. Такой вызов работает, как если бы он был разделён на 2 строчки:

```
f = obj.go; // вычисляется выражение
(переменная f ссылается на код функции)
f();        // вызов функции, на которую
ссылается f
```

Здесь `f()` выполняется как функция, без передачи значения `this`.

4. Тут похожая ситуация на случай `(3)` – идёт потеря значения `this`.

Чтобы объяснить поведение в примерах `(3)` и `(4)`, нам нужно помнить, что доступ к свойству (через точку или квадратные скобки) возвращает специальное значение ссылочного типа (Reference Type).

За исключением вызова метода, любая другая операция (подобно операции присваивания `=` или сравнения через логические операторы, например `||`) превращает это значение в обычное, которое не несёт информации, позволяющей установить `this`.

К условию

Использование "this" в литерале объекта

Ответ: ошибка.

Проверьте:

```
function makeUser() {
  return {
    name: "Джон",
    ref: this
  };
};

let user = makeUser();

alert( user.ref.name ); // Error: Cannot read
property 'name' of undefined
```

Это потому, что правила, которые определяют значение `this`, никак не смотрят на объявление объекта. Важен лишь момент вызова метода.

Здесь значение `this` внутри `makeUser()` является `undefined`, потому что `makeUser()` вызвана как функция, не через «точку» как метод.

Литерал объекта сам по себе не влияет на `this`. Значение `this` одно для всей функции и блоков кода в ней, литеральные объекты не меняют его.

Таким образом, при создании объекта `ref: this` берёт текущее значение `this` функции `makeUser()`.

А вот противоположный случай:

```
function makeUser() {
  return {
    name: "Джон",
```

```
  };
};

let user = makeUser();

alert( user.ref().name ); // Джон
```

```
ref() {
  return this;
}
```

Теперь это работает, поскольку `user.ref()` вызывается как метод. И значением `this` становится объект перед точкой `.`.

К условию

Создайте калькулятор

```
let calculator = {
  sum() {
    return this.a + this.b;
  },

  mul() {
    return this.a * this.b;
  },

  read() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  }
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Открыть решение с тестами в песочнице.

К условию

Цепь вызовов

Решением является возврат самого объекта в каждом методе.

```
let ladder = {
  step: 0,
  up() {
    this.step++;

  },
  down() {
    this.step--;

  },
  showStep() {
    alert( this.step );

  }
}

ladder.up().up().down().up().down().showStep(); //
1
```

```
return this;
return this;
return this;
```

Мы также можем писать один вызов на одной строке. Для длинной цепи вызовов это более читабельно:

```
ladder
  .up()
  .up()
  .down()
  .up()
  .down()
  .showStep(); // 1
```

Открыть решение с тестами в песочнице.

К условию

Две функции - один объект

Да, возможно.

Если функция возвращает объект, то вместо `this` будет возвращён этот объект.

Например, они могут вернуть один и тот же объект `obj`, определённый снаружи:

```
let obj = {};

function A() { return obj; }
function B() { return obj; }

alert( new A() == new B() ); // true
```

К условию

Создание калькулятора при помощи конструктора

```
function Calculator() {

  this.read = function() {
    this.a = +prompt('a?', 0);
    this.b = +prompt('b?', 0);
  };

  this.sum = function() {
```

```
    return this.a + this.b;
  };

  this.mul = function() {
    return this.a * this.b;
  };
}

let calculator = new Calculator();
calculator.read();

alert( "Sum=" + calculator.sum() );
alert( "Mul=" + calculator.mul() );
```

Открыть решение с тестами в песочнице.

К условию

Создаём Accumulator

```
function Accumulator(startingValue) {
  this.value = startingValue;

  this.read = function() {
    this.value += +prompt('Сколько нужно
добавить?', 0);
  };

}

let accumulator = new Accumulator(1);
accumulator.read();
accumulator.read();
alert(accumulator.value);
```

Открыть решение с тестами в песочнице.

К условию

Можно ли добавить свойство строке?

Попробуйте запустить код:

```
let str = "Привет";

str.test = 5; // (*)

alert(str.test);
```

В зависимости от того, используете ли вы строгий режим (`use strict`) или нет, результат может быть:

1. `undefined` (без strict)
2. Ошибка (strict mode)

Почему? Давайте посмотрим что происходит в строке кода, отмеченной `(*)`:

1. В момент обращения к свойству `str` создаётся «объект-обёртка».
2. В строгом режиме, попытка изменения этого объекта выдаёт ошибку.
3. Без строгого режима, операция продолжается, объект получает свойство `test`, но после этого он удаляется, так что на последней линии `str` больше не имеет свойства `test`.

Данный пример наглядно показывает, что примитивы не являются объектами.

Они не могут хранить дополнительные данные.

К условию

Числа

Сумма пользовательских чисел

```
let a = +prompt("Введите первое число", "");
let b = +prompt("Введите второе число", "");

alert( a + b );
```

Обратите внимание, что мы использовали унарный оператор `+` перед `prompt`, он преобразует значение в числовой формат.

В противном случае, `a` и `b` будут строками, и после суммирования произойдёт конкатенация двух строк, а именно: `"1" + "2" = "12"`.

К условию

Почему 6.35.toFixed(1) == 6.3?

Во внутреннем двоичном представлении 6.35 является бесконечной двоичной дробью. Хранится она с потерей точности…

Давайте посмотрим:

```
alert( 6.35.toFixed(20) ); //
6.34999999999999964473
```

Потеря точности может как увеличивать, так и уменьшать число. В данном случае число становится чуть меньше, поэтому оно округляется в меньшую сторону.

А для числа `1.35`?

```
alert( 1.35.toFixed(20) ); //
1.35000000000000008882
```

Тут потеря точности приводит к увеличению числа, поэтому округление произойдёт в большую сторону.

Каким образом можно исправить ошибку в округлении числа `6.35`?

Мы должны приблизить его к целому числу, перед округлением:

```
alert( (6.35 * 10).toFixed(20) ); //
63.50000000000000000000
```

Обратите внимание, что для числа `63.5` не происходит потери точности. Дело в том, что десятичная часть `0.5` на самом деле `1/2`.

Дробные числа, делённые на степень `2`, точно представлены в двоичной системе, теперь мы можем округлить число:

```
alert( Math.round(6.35 * 10) / 10); // 6.35 -> 63.5
-> 64(rounded) -> 6.4
```

К условию

Ввод числового значения

```
function readNumber() {
  let num;

  do {
    num = prompt("Введите число", 0);
  } while ( !isFinite(num) );

  if (num === null || num === '') return null;

  return +num;
}

alert(`Число: ${readNumber()}`);
```

Решение немного сложнее, чем могло бы быть, потому что нам надо обрабатывать `null` и пустую строку.

Следовательно, запрашиваем ввод числового значения, пока посетитель его не введёт. И `null` (отмена) и пустая строка также соответствуют данному условию, потому что при приведении к числу они равны `0`.

После того, как цикл завершится, нам нужно проверить введённое значение на `null` и пустую строку (вернуть `null`), потому что после

преобразования `null` в число, функция вернёт `0`.

Открыть решение с тестами в песочнице.

К условию

Бесконечный цикл по ошибке

Потому что `i` никогда не станет равным `10`.

Запустите скрипт и вы увидите *реальные* значения `i`:

```
let i = 0;
while (i < 11) {
  i += 0.2;
  if (i > 9.8 && i < 10.2) alert( i );
}
```

Ни одно из этих чисел не равно `10`.

Это происходит из-за потери точности, при прибавлении таких дробей как `0.2`.

Вывод: избегайте проверок на равенство при работе с десятичными дробями.

К условию

Случайное число от min до max

Нам нужно преобразовать каждое значение из

интервала 0…1 в значения от `min` до `max`.

Это можно сделать в 2 шага:

1. Если мы умножим случайное число от 0…1 на `max-min`, тогда интервал возможных значений от `0..1` увеличивается до `0..max-min`.
2. И, если мы прибавим `min`, то интервал станет от `min` до `max`.

Функция:

```
function random(min, max) {
  return min + Math.random() * (max - min);
}

alert( random(1, 5) );
alert( random(1, 5) );
alert( random(1, 5) );
```

К условию

Случайное целое число от min до max

Простое, но неправильное решение

Самое простое, но неправильное решение – генерировать случайное число от `min` до `max` и округлять его:

```
function randomInteger(min, max) {
  let rand = min + Math.random() * (max - min);
  return Math.round(rand);
}
```

```
alert( randomInteger(1, 3) );
```

Функция будет работать, но неправильно. Вероятность получить `min` и `max` значения в 2 раза меньше, чем любое другое число.

Если вы запустите приведённый выше пример, то заметите, что `2` появляется чаще всего.

Это происходит потому, что метод `Math.round()` получает случайные числа из интервала `1..3` и округляет их следующим образом:

```
число от 1    ... до 1.4999999999  округлится до 1
число от 1.5  ... до 2.4999999999  округлится до 2
число от 2.5  ... до 2.9999999999  округлится до 3
```

Теперь становится понятно, что `1` получает в 2 раза меньше значений, чем `2`. То же самое с `3`.

Правильное решение задачи

Есть много правильных решений этой задачи. Одно из них – правильно указать границы интервала. Чтобы выровнять интервалы, мы можем генерировать числа от `0.5 до 3.5`, это позволит добавить *необходимые вероятности* к `min` и `max`:

```
alert( randomInteger(1, 3) );
function randomInteger(min, max) {
  // получить случайное число от
```

```
(min-0.5) до (max+0.5)
  let rand = min - 0.5 + Math.random()
* (max - min + 1);
  return Math.round(rand);
}
```

Другое правильное решение – это использовать `Math.floor` для получения случайного числа от `min` до `max+1`:

```
alert( randomInteger(1, 3) );
function randomInteger(min, max) {
  // случайное число от min до (max+1)
  let rand = min + Math.random() *
(max + 1 - min);
  return Math.floor(rand);
}
```

Теперь все интервалы отображаются следующим образом:

```
число от 1  ... до 1.9999999999  округлится до 1
число от 2  ... до 2.9999999999  округлится до 2
число от 3  ... до 3.9999999999  округлится до 3
```

Все интервалы имеют одинаковую длину, что выравнивает вероятность получения случайных чисел.

К условию

Сделать первый символ заглавным

Мы не можем просто заменить первый символ, так как строки в JavaScript неизменяемы.

Но можно пересоздать строку на основе существующей, с заглавным первым символом:

```
let newStr = str[0].toUpperCase() + str.slice(1);
```

Однако есть небольшая проблемка. Если строка пуста, `str[0]` вернёт `undefined`, а у `undefined` нет метода `toUpperCase()`, поэтому мы получим ошибку.

Выхода два:

1. Использовать `str.charAt(0)`, поскольку этот метод всегда возвращает строку (для пустой строки — пустую).
2. Добавить проверку на пустую строку.

Вот второй вариант:

```
function ucFirst(str) {
  if (!str) return str;

  return str[0].toUpperCase() + str.slice(1);
```

```
}

alert( ucFirst("вася") ); // Вася
```

Открыть решение с тестами в песочнице.

К условию

Проверка на спам

Для поиска без учёта регистра символов переведём всю строку в нижний регистр, а потом проверим, есть ли в ней искомые подстроки:

```
function checkSpam(str) {
  let lowerStr = str.toLowerCase();

  return lowerStr.includes('viagra') ||
lowerStr.includes('xxx');
}

alert( checkSpam('buy ViAgRA now') );
alert( checkSpam('free xxxxx') );
alert( checkSpam("innocent rabbit") );
```

Открыть решение с тестами в песочнице.

К условию

Усечение строки

Строка, которую мы возвращаем, должна быть не длиннее `maxlength`, поэтому, если мы обрезаем строку, то мы должны убрать на один символ больше, чем `maxlength` — чтобы хватило места на многоточие.

Имейте в виду, что в качестве многоточия здесь используется … — ровно один специальный юникодный символ. Это не то же самое, что `...` — три точки.

```
function truncate(str, maxlength) {
  return (str.length > maxlength) ?
    str.slice(0, maxlength - 1) + '…' : str;
}
```

Открыть решение с тестами в песочнице.

К условию

Выделить число

```
function extractCurrencyValue(str) {
  return +str.slice(1);
}
```

Открыть решение с тестами в песочнице.

К условию

Массивы

Скопирован ли массив?

Выведется `4`:

```
let fruits = ["Яблоки", "Груша", "Апельсин"];

let shoppingCart = fruits;

shoppingCart.push("Банан");

alert( fruits.length ); // 4
```

Потому, что массивы – это объекты. Обе переменные `shoppingCart` и `fruits` являются ссылками на один и тот же массив.

К условию

Операции с массивами

```
let styles = ["Джаз", "Блюз"];
styles.push("Рок-н-ролл");
styles[Math.floor((styles.length - 1) / 2)] =
"Классика";
alert( styles.shift() );
styles.unshift("Рэп", "Регги");
```

К условию

Вызов в контексте массива

Вызов `arr[2]()` синтаксически – старый добрый `obj[method]()`, в роли `obj` – `arr`, а в роли `method` – `2`.

Итак, у нас есть вызов функции `arr[2]` как метода объекта. Соответственно, он получает в качестве `this` объект `arr` и выводит массив:

```
let arr = ["a", "b"];
```

```
arr.push(function() {
  alert( this );
})

arr[2](); // "a","b",function
```

У массива в итоге 3 элемента: сначала их было 2, плюс функция.

К условию

Сумма введённых чисел

Обратите внимание на малозаметную, но важную деталь решения. Мы не преобразуем `value` в число сразу после `prompt`, потому что после `value = +value` мы не сможем отличить пустую строку (конец записи) от «0» (разрешённое число). Мы сделаем это позже.

```
function sumInput() {

  let numbers = [];

  while (true) {

    let value = prompt("Введите число", 0);

    // Прекращаем ввод?
    if (value === "" || value === null || !
isFinite(value)) break;

    numbers.push(+value);
  }

  let sum = 0;
  for (let number of numbers) {
    sum += number;
  }
  return sum;
}
```

```
alert( sumInput() );
```

К условию

Подмассив наибольшей суммы

Медленное решение

Можно посчитать все возможные подсуммы.

Самый простой путь – посчитать суммы подмассивов, начиная с каждого элемента по очереди.

Например, для `[-1, 2, 3, -9, 11]`:

```
// Начиная с -1:
-1
-1 + 2
-1 + 2 + 3
-1 + 2 + 3 + (-9)
-1 + 2 + 3 + (-9) + 11

// Начиная с 2:
2
2 + 3
2 + 3 + (-9)
2 + 3 + (-9) + 11

// Начиная с 3:
3
3 + (-9)
3 + (-9) + 11

// Начиная с -9
-9
-9 + 11

// Начиная с 11
11
```

Реализуется с помощью вложенного цикла: внешний цикл проходит по элементам массива, а внутренний считает подсумму, начиная с текущего элемента.

```
function getMaxSubSum(arr) {
  let maxSum = 0; // если элементов не будет –
возвращаем 0

  for (let i = 0; i < arr.length; i++) {
    let sumFixedStart = 0;
    for (let j = i; j < arr.length; j++) {
      sumFixedStart += arr[j];
      maxSum = Math.max(maxSum, sumFixedStart);
    }
  }

  return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
```

Это решение имеет оценку сложности $O(n^2)$. Другими словами, если мы увеличим размер массива в 2 раза, время выполнения алгоритма увеличится в 4 раза.

Для больших массивов(1000, 10000 или больше элементов) такие алгоритмы могут приводить к серьёзным «тормозам».

Быстрое решение

Идём по массиву и накапливаем текущую частичную сумму элементов в переменной `s`. Если `s` в какой-то момент становится

отрицательной – присваиваем `s=0`.
Максимальный из всех `s` и будет ответом.

Если объяснение недостаточно понятно, посмотрите на код, он вполне лаконичен:

```
function getMaxSubSum(arr) {
  let maxSum = 0;
  let partialSum = 0;

  for (let item of arr) { // для каждого элемента массива
    partialSum += item; // добавляем значение элемента к partialSum
    maxSum = Math.max(maxSum, partialSum); // запоминаем максимум на данный момент
    if (partialSum < 0) partialSum = 0; // ноль если отрицательное
  }

  return maxSum;
}

alert( getMaxSubSum([-1, 2, 3, -9]) ); // 5
alert( getMaxSubSum([-1, 2, 3, -9, 11]) ); // 11
alert( getMaxSubSum([-2, -1, 1, 2]) ); // 3
alert( getMaxSubSum([100, -9, 2, -3, 5]) ); // 100
alert( getMaxSubSum([1, 2, 3]) ); // 6
alert( getMaxSubSum([-1, -2, -3]) ); // 0
```

Этот алгоритм требует ровно 1 проход по массиву и его оценка сложности O(n).

Больше информации об алгоритме тут: Задача поиска максимальной суммы подмассива. Если всё ещё не очевидно как это работает, просмотрите алгоритм в примерах выше, это будет лучше всяких слов.

Открыть решение с тестами в песочнице.

К условию

Методы массивов

Переведите текст вида border-left-width в borderLeftWidth

```
function camelize(str) {
  return str
    .split('-') // разбивает 'my-long-word' на
массив ['my', 'long', 'word']
    .map(
      // Переводит в верхний регистр первые буквы
всех элементом массива за исключением первого
      // превращает ['my', 'long', 'word'] в ['my',
'Long', 'Word']
      (word, index) => index == 0 ? word :
word[0].toUpperCase() + word.slice(1)
    )
    .join(''); // соединяет ['my', 'Long', 'Word']
в 'myLongWord'
}
```

Открыть решение с тестами в песочнице.

К условию

Фильтрация по диапазону

```
function filterRange(arr, a, b) {
  // добавлены скобки вокруг выражения для
улучшения читабельности
  return arr.filter(item => (a <= item && item <=
b));
}
```

```
let arr = [5, 3, 8, 1];

let filtered = filterRange(arr, 1, 4);

alert( filtered ); // 3,1 (совпадающие значения)

alert( arr ); // 5,3,8,1 (без изменений)
```

Открыть решение с тестами в песочнице.

К условию

Фильтрация по диапазону "на месте"

```
function filterRangeInPlace(arr, a, b) {

  for (let i = 0; i < arr.length; i++) {
    let val = arr[i];

    // удалить, если за пределами интервала
    if (val < a || val > b) {
      arr.splice(i, 1);
      i--;
    }
  }

}

let arr = [5, 3, 8, 1];

filterRangeInPlace(arr, 1, 4); // удалены числа вне
диапазона 1..4

alert( arr ); // [3, 1]
```

Открыть решение с тестами в песочнице.

К условию

Сортировать в порядке по

убыванию

```
let arr = [5, 2, 1, -10, 8];

arr.sort((a, b) => b - a);

alert( arr );
```

К условию

Скопировать и отсортировать массив

Для копирования массива используем `slice()` и тут же – сортировку:

```
function copySorted(arr) {
  return arr.slice().sort();
}

let arr = ["HTML", "JavaScript", "CSS"];


alert( sorted );
alert( arr );
let sorted = copySorted(arr);
```

К условию

Создать расширяемый калькулятор

Обратите внимание, как хранятся методы. Они просто добавляются к внутреннему объекту.

Все тесты и числовые преобразования выполняются в методе `calculate`. В будущем он может быть расширен для поддержки более сложных выражений.

```
function Calculator() {

  this.methods = {
    "-": (a, b) => a - b,
    "+": (a, b) => a + b
  };

  this.calculate = function(str) {

    let split = str.split(' '),
      a = +split[0],
      op = split[1],
      b = +split[2]

    if (!this.methods[op] || isNaN(a) || isNaN(b))
{
      return NaN;
    }

    return this.methods[op](a, b);
  }

  this.addMethod = function(name, func) {
    this.methods[name] = func;
  };
}
```

```
function Calculator() {

  this.methods = {
    "-": (a, b) => a - b,
    "+": (a, b) => a + b
  };

  this.calculate = function(str) {

    let split = str.split(' '),
      a = +split[0],
      op = split[1],
      b = +split[2]

    if (!this.methods[op] || isNaN(a) || isNaN(b))
{
      return NaN;
```

```
    }

    return this.methods[op](a, b);
  }

  this.addMethod = function(name, func) {
    this.methods[name] = func;
  };
}
```

Открыть решение с тестами в песочнице.

К условию

Трансформировать в массив имён

```
let vasya = { name: "Вася", age: 25 };
let petya = { name: "Петя", age: 30 };
let masha = { name: "Маша", age: 28 };

let users = [ vasya, petya, masha ];

let names = users.map(item => item.name);

alert( names ); // Вася, Петя, Маша
```

К условию

Трансформировать в объекты

```
let vasya = { name: "Вася", surname: "Пупкин", id:
1 };
let petya = { name: "Петя", surname: "Иванов", id:
2 };
let masha = { name: "Маша", surname: "Петрова", id:
3 };

let users = [ vasya, petya, masha ];
```

```
/*
usersMapped = [
  { fullName: "Вася Пупкин", id: 1 },
  { fullName: "Петя Иванов", id: 2 },
  { fullName: "Маша Петрова", id: 3 }
]
*/

alert( usersMapped[0].id ); // 1
alert( usersMapped[0].fullName ); // Вася Пупкин
let usersMapped = users.map(user => ({
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));
```

Обратите внимание, что для стрелочных функций мы должны использовать дополнительные скобки.

Мы не можем написать вот так:

```
let usersMapped = users.map(user =>
  fullName: `${user.name} ${user.surname}`,
  id: user.id
});
{
```

Как мы помним, есть две функции со стрелками: без тела `value => expr` и с телом `value => {...}`.

Здесь JavaScript будет трактовать `{` как начало тела функции, а не начало объекта. Чтобы обойти это, нужно заключить их в «нормальные» скобки:

```
let usersMapped = users.map(user =>
  fullName: `${user.name} ${user.surname}`,
  id: user.id
}));
```

```
(
{
```

Теперь всё хорошо.

К условию

Отсортировать пользователей по возрасту

```
function sortByAge(arr) {
  arr.sort((a, b) => a.age > b.age ? 1 : -1);
}

let vasya = { name: "Вася", age: 25 };
let petya = { name: "Петя", age: 30 };
let masha = { name: "Маша", age: 28 };

let arr = [ vasya, petya, masha ];

sortByAge(arr);

// теперь отсортировано: [vasya, masha, petya]
alert(arr[0].name); // Вася
alert(arr[1].name); // Маша
alert(arr[2].name); // Петя
```

К условию

Перемешайте массив

Простым решением может быть:

```
let arr = [1, 2, 3];
shuffle(arr);
alert(arr);
function shuffle(array) {
```

```
  array.sort(() => Math.random() -
0.5);
}
```

Это, конечно, будет работать, потому что `Math.random() - 0.5` отдаёт случайное число, которое может быть положительным или отрицательным, следовательно, функция сортировки меняет порядок элементов случайным образом.

Но поскольку метод `sort` не предназначен для использования в таких случаях, не все возможные варианты имеют одинаковую вероятность.

Например, рассмотрим код ниже. Он запускает `shuffle` 1000000 раз и считает вероятность появления для всех возможных вариантов `arr`:

```
function shuffle(array) {
  array.sort(() => Math.random() - 0.5);
}

// подсчёт вероятности для всех возможных вариантов
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// показать количество всех возможных вариантов
for (let key in count) {
```

```
  alert(`${key}: ${count[key]}`);
}
```

Результат примера (зависят от движка JS):

```
123: 250706
132: 124425
213: 249618
231: 124880
312: 125148
321: 125223
```

Теперь мы отчётливо видим допущенное отклонение: `123` и `213` появляются намного чаще, чем остальные варианты.

Результаты этого кода могут варьироваться при запуске на разных движках JavaScript, но очевидно, что такой подход не надёжен.

Так почему это не работает? Если говорить простыми словами, то `sort` это «чёрный ящик»: мы бросаем в него массив и функцию сравнения, ожидая получить отсортированный массив. Но из-за абсолютной хаотичности сравнений чёрный ящик сходит с ума, и как именно он сходит с ума, зависит от конкретной его реализации, которая различна в разных движках JavaScript.

Есть и другие хорошие способы решить эту задачу. Например, есть отличный алгоритм под названием Тасование Фишера — Йетса. Суть заключается в том, чтобы проходить по массиву в обратном порядке и менять местами каждый элемент со случайным элементом, который

находится перед ним.

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1)); //
случайный индекс от 0 до i

    // поменять элементы местами
    // мы используем для этого синтаксис
"деструктурирующее присваивание"
    // подробнее о нём - в следующих главах
    // то же самое можно записать как:
    // let t = array[i]; array[i] = array[j];
array[j] = t
    [array[i], array[j]] = [array[j], array[i]];
  }
}
```

Давайте проверим эту реализацию на том же примере:

```
function shuffle(array) {
  for (let i = array.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1));
    [array[i], array[j]] = [array[j], array[i]];
  }
}

// подсчёт вероятности для всех возможных вариантов
let count = {
  '123': 0,
  '132': 0,
  '213': 0,
  '231': 0,
  '321': 0,
  '312': 0
};

for (let i = 0; i < 1000000; i++) {
  let array = [1, 2, 3];
  shuffle(array);
  count[array.join('')]++;
}

// показать количество всех возможных вариантов
for (let key in count) {
  alert(`${key}: ${count[key]}`);
}
```

Пример вывода:

```
123: 166693
132: 166647
213: 166628
231: 167517
312: 166199
321: 166316
```

Теперь всё в порядке: все варианты появляются с одинаковой вероятностью.

Кроме того, если посмотреть с точки зрения производительности, то алгоритм «Тасование Фишера — Йетса» намного быстрее, так как в нём нет лишних затрат на сортировку.

К условию

Получить средний возраст

```
function getAverageAge(users) {
  return users.reduce((prev, user) => prev +
user.age, 0) / users.length;
}

let vasya = { name: "Вася", age: 25 };
let petya = { name: "Петя", age: 30 };
let masha = { name: "Маша", age: 29 };

let arr = [ vasya, petya, masha ];

alert( getAverageAge(arr) ); // 28
```

К условию

Оставить уникальные элементы массива

Давайте пройдёмся по элементам массива:

Для каждого элемента мы проверим, есть ли он в массиве с результатом.

Если есть, то игнорируем его, а если нет – добавляем к результатам.

```
function unique(arr) {
  let result = [];

  for (let str of arr) {
    if (!result.includes(str)) {
      result.push(str);
    }
  }

  return result;
}

let strings = ["кришна", "кришна", "харе", "харе",
  "харе", "харе", "кришна", "кришна", ":-O"
];

alert( unique(strings) ); // кришна, харе, :-O
```

Код работает, но в нём есть потенциальная проблема с производительностью.

Метод `result.includes(str)` внутри себя обходит массив `result` и сравнивает каждый элемент с `str`, чтобы найти совпадение.

Таким образом, если `result` содержит `100` элементов и ни один не совпадает со `str`, тогда он обойдёт весь `result` и сделает ровно `100` сравнений. А если `result` большой, например, `10000`, то будет произведено `10000` сравнений.

Само по себе это не проблема, потому что

движки JavaScript очень быстрые, поэтому обход `10000` элементов массива занимает считанные микросекунды.

Но мы делаем такую проверку для каждого элемента `arr` в цикле `for`.

Поэтому, если `arr.length` равен `10000`, у нас будет что-то вроде `10000*10000` = 100 миллионов сравнений. Это многовато.

Вот почему данное решение подходит только для небольших массивов.

Далее в главе Map и Set мы увидим, как его оптимизировать.

Открыть решение с тестами в песочнице.

К условию

Map и Set

Фильтрация уникальных элементов массива

```
function unique(arr) {
  return Array.from(new Set(arr));
}
```

Открыть решение с тестами в песочнице.

К условию

Отфильтруйте анаграммы

Чтобы найти все анаграммы, давайте разобьём каждое слово на буквы и отсортируем их, а потом объединим получившийся массив снова в строку. После этого все анаграммы будут одинаковы.

Например:

```
nap, pan -> anp
ear, era, are -> aer
cheaters, hectares, teachers -> aceehrst
...
```

Мы будем использовать отсортированные строки как ключи в коллекции Map, для того чтобы сопоставить каждому ключу только одно значение:

```
function aclean(arr) {
  let map = new Map();

  for (let word of arr) {
    // разбиваем слово на буквы, сортируем и
объединяем снова в строку

    map.set(sorted, word);
  }

  return Array.from(map.values());
}

let arr = ["nap", "teachers", "cheaters", "PAN",
"ear", "era", "hectares"];
```

```
alert( aclean(arr) );
    let sorted =
word.toLowerCase().split("").sort().join("");
// (*)
```

Строка с отсортированными буквами получается в результате цепочки вызовов в строке `(*)`.

Для удобства, давайте разделим это на несколько строк:

```
let sorted = arr[i] // PAN
  .toLowerCase() // pan
  .split("") // ["p","a","n"]
  .sort() // ["a","n","p"]
  .join(""); // anp
```

Два разных слова `'PAN'` и `'nap'` принимают ту же самую форму после сортировки букв – `'anp'`.

Следующая линия помещает слово в объект Map:

```
map.set(sorted, word);
```

Если мы когда-либо ещё встретим слово в той же отсортированной форме, тогда это слово перезапишет значение с тем же ключом в объекте. Таким образом, нескольким словам у нас будет всегда соответствовать одна отсортированная форма.

В конце `Array.from(map.values())` принимает итерируемый объект значений объекта Map (в данном случае нам не нужны

ключи) и возвращает их в виде массива.

Также в этом случае вместо `Map` мы можем использовать простой объект, потому что ключи являются строками.

В этом случае решение может выглядеть так:

```
function aclean(arr) {
  let obj = {};

  for (let i = 0; i < arr.length; i++) {
    let sorted =
arr[i].toLowerCase().split("").sort().join("");
    obj[sorted] = arr[i];
  }

  return Object.values(obj);
}

let arr = ["nap", "teachers", "cheaters", "PAN",
"ear", "era", "hectares"];

alert( aclean(arr) );
```

Открыть решение с тестами в песочнице.

К условию

Перебираемые ключи

Это потому что `map.keys()` возвращает итерируемый объект, а не массив.

Мы можем конвертировать его в массив с помощью `Array.from`:

```
let map = new Map();
```

```
map.set("name", "John");


keys.push("more");

alert(keys); // name, more
let keys = Array.from(map.keys());
```

К условию

WeakMap и WeakSet

Хранение отметок "не прочитано"

Можно хранить прочитанные сообщения в `WeakSet`:

```
let messages = [
    {text: "Hello", from: "John"},
    {text: "How goes?", from: "John"},
    {text: "See you soon", from: "Alice"}
];

let readMessages = new WeakSet();

// Два сообщения были прочитаны
readMessages.add(messages[0]);
readMessages.add(messages[1]);
// readMessages содержит 2 элемента

// ...давайте снова прочитаем первое сообщение!
readMessages.add(messages[0]);
// readMessages до сих пор содержит 2 элемента

// Вопрос: было ли сообщение message[0] прочитано?
alert("Read message 0: " +
readMessages.has(messages[0])); // true
```

```
messages.shift();
// теперь readMessages содержит 1 элемент (хотя
технически память может быть очищена позже)
```

`WeakSet` позволяет хранить набор сообщений и легко проверять наличие сообщения в нём.

Он очищается автоматически. Минус в том, что мы не можем перебрать его содержимое, не можем получить «все прочитанные сообщения» напрямую. Но мы можем сделать это, перебирая все сообщения и фильтруя те, которые находятся в `WeakSet`.

Альтернативным решением может быть добавление свойства вида `message.isRead=true` к сообщению после его прочтения. Так как сообщения принадлежат чужому коду, то это не очень хорошо, но если использовать свойство-символ, то вероятность конфликтов будет небольшой.

Например:

```
// символьное свойство вместо имени, которое
известно только нашему коду
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

Теперь чужой код вряд ли увидит наше дополнительное свойство.

Хотя символы и позволяют уменьшить вероятность проблем, использование здесь `WeakSet` лучше с архитектурной точки зрения.

К условию

Хранение времени прочтения

Для хранения даты мы можем использовать `WeakMap`:

```
let messages = [
  {text: "Hello", from: "John"},
  {text: "How goes?", from: "John"},
  {text: "See you soon", from: "Alice"}
];

let readMap = new WeakMap();

readMap.set(messages[0], new Date(2017, 1, 1));
// Объект Date мы рассмотрим позднее
```

К условию

Object.keys, values, entries

Сумма свойств объекта

```
function sumSalaries(salaries) {

  let sum = 0;
  for (let salary of Object.values(salaries)) {
    sum += salary;
  }

  return sum; // 650
}

let salaries = {
```

```
  "John": 100,
  "Pete": 300,
  "Mary": 250
};

alert( sumSalaries(salaries) ); // 650
```

Или, как вариант, мы можем получить сумму, используя методы `Object.values` и `reduce`:

```
// reduce перебирает массив значений salaries,
// складывает их
// и возвращает результат
function sumSalaries(salaries) {
  return Object.values(salaries).reduce((a, b) => a
+ b, 0) // 650
}
```

Открыть решение с тестами в песочнице.

К условию

Подсчёт количества свойств объекта

```
function count(obj) {
  return Object.keys(obj).length;
}
```

Открыть решение с тестами в песочнице.

К условию

Деструктурирующее присваивание

Деструктурирующее присваивание

```
let user = {
  name: "John",
  years: 30
};

let {name, years: age, isAdmin = false} = user;

alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

К условию

Максимальная зарплата

```
function topSalary(salaries) {

  let max = 0;
  let maxName = null;

  for(const [name, salary] of
Object.entries(salaries)) {
    if (max < salary) {
      max = salary;
      maxName = name;
    }
  }

  return maxName;
}
```

Открыть решение с тестами в песочнице.

К условию

Создайте дату

Конструктор `new Date` стандартно использует местную временную зону. Единственная важная вещь, которую нужно запомнить – это то, что месяцы начинаются с нуля.

Поэтому февраль обозначается числом 1.

```
let d = new Date(2012, 1, 20, 3, 12);
alert( d );
```

К условию

Покажите день недели

Метод `date.getDay()` возвращает номер дня недели, начиная с воскресенья.

Создадим массив дней недели, чтобы получить имя нужного дня по его номеру:

```
function getWeekDay(date) {
  let days = ['ВС', 'ПН', 'ВТ', 'СР', 'ЧТ', 'ПТ',
'СБ'];

  return days[date.getDay()];
}

let date = new Date(2014, 0, 3); // 3 января 2014
года
```

```
alert( getWeekDay(date) ); // ПТ
```

Открыть решение с тестами в песочнице.

К условию

День недели в европейской нумерации

```
function getLocalDay(date) {

  let day = date.getDay();

  if (day == 0) { // день недели 0 (воскресенье) в европейской нумерации будет 7
    day = 7;
  }

  return day;
}
```

Открыть решение с тестами в песочнице.

К условию

Какой день месяца был много дней назад?

Идея проста: нужно вычесть заданное количество дней из `date`:

```
function getDateAgo(date, days) {
  date.setDate(date.getDate() - days);
  return date.getDate();
}
```

...Но функция не должна изменять объект `date`.

Это очень важно, поскольку внешний код, передающий нам объект, не ожидает его изменения.

Это можно осуществить путём клонирования даты:

```
function getDateAgo(date, days) {
  let dateCopy = new Date(date);

  dateCopy.setDate(date.getDate() - days);
  return dateCopy.getDate();
}

let date = new Date(2015, 0, 2);

alert( getDateAgo(date, 1) ); // 1, (1 Jan 2015)
alert( getDateAgo(date, 2) ); // 31, (31 Dec 2014)
alert( getDateAgo(date, 365) ); // 2, (2 Jan 2014)
```

Открыть решение с тестами в песочнице.

К условию

Последнее число месяца?

Создадим дату из следующего месяца, но в день передадим 0:

```
function getLastDayOfMonth(year, month) {
  let date = new Date(year, month + 1, 0);
  return date.getDate();
}

alert( getLastDayOfMonth(2012, 0) ); // 31
alert( getLastDayOfMonth(2012, 1) ); // 29
alert( getLastDayOfMonth(2013, 1) ); // 28
```

Обычно даты начинаются с 1, но технически возможно передать любое число, и дата сама

себя поправит. Так что если передать 0, то это значение будет соответствовать «один день перед первым числом месяца», другими словами: «последнее число прошлого месяца».

Открыть решение с тестами в песочнице.

К условию

Сколько сегодня прошло секунд?

Чтобы получить количество секунд, нужно сгенерировать объект `date` на самое начало текущего дня – 00:00:00, а затем вычесть полученное значение из «сейчас».

Разность даст нам количество миллисекунд с начала дня, делим его на 1000 и получаем секунды:

```
function getSecondsToday() {
  let now = new Date();

  // создаём объект с текущими днём/месяцем/годом
  let today = new Date(now.getFullYear(),
now.getMonth(), now.getDate());

  let diff = now - today; // разница в
миллисекундах
  return Math.round(diff / 1000); // получаем
секунды
}

alert( getSecondsToday() );
```

В качестве альтернативного решения можно получить часы/минуты и преобразовать их в

секунды:

```
function getSecondsToday() {
  let d = new Date();
  return d.getHours() * 3600 + d.getMinutes() * 60
+ d.getSeconds();
}
```

К условию

Сколько секунд осталось до завтра?

Чтобы получить количество миллисекунд до завтра, можно из «завтра 00:00:00» вычесть текущую дату.

Сперва сгенерируем дату на «завтра» и сделаем следующее:

```
function getSecondsToTomorrow() {
  let now = new Date();

  // завтрашняя дата
  let tomorrow = new Date(now.getFullYear(),
now.getMonth(), );

  let diff = tomorrow - now; // разница в
миллисекундах
  return Math.round(diff / 1000); // преобразуем в
секунды
}
now.getDate()+1
```

Альтернативное решение:

```
function getSecondsToTomorrow() {
  let now = new Date();
  let hour = now.getHours();
  let minutes = now.getMinutes();
```

```
  let seconds = now.getSeconds();
  let totalSecondsToday = (hour * 60 + minutes) *
60 + seconds;
  let totalSecondsInADay = 86400;

  return totalSecondsInADay - totalSecondsToday;
}
```

Учтите, что многие страны переходят с зимнего времени на летнее и обратно, так что могут быть дни длительностью в 23 или 25 часов. Такие дни, если это важно, можно обрабатывать отдельно.

К условию

Форматирование относительной даты

Чтобы получить время с `date` по текущий момент, нужно вычесть даты.

```
function formatDate(date) {
  let diff = new Date() - date; // разница в
миллисекундах

  if (diff < 1000) { // меньше 1 секунды
    return 'прямо сейчас';
  }

  let sec = Math.floor(diff / 1000); //
преобразовать разницу в секунды

  if (sec < 60) {
    return sec + ' сек. назад';
  }

  let min = Math.floor(diff / 60000); //
преобразовать разницу в минуты
  if (min < 60) {
    return min + ' мин. назад';
  }
```

```
  // отформатировать дату
  // добавить ведущие нули к единственной цифре
дню/месяцу/часам/минутам
  let d = date;
  d = [
    '0' + d.getDate(),
    '0' + (d.getMonth() + 1),
    '' + d.getFullYear(),
    '0' + d.getHours(),
    '0' + d.getMinutes()
  ].map(component => component.slice(-2)); // взять
последние 2 цифры из каждой компоненты

  // соединить компоненты в дату
  return d.slice(0, 3).join('.') + ' ' +
d.slice(3).join(':');
}

alert( formatDate(new Date(new Date - 1)) ); //
"прямо сейчас"

alert( formatDate(new Date(new Date - 30 * 1000))
); // "30 сек. назад"

alert( formatDate(new Date(new Date - 5 * 60 *
1000)) ); // "5 мин. назад"

// вчерашняя дата вроде 31.12.2016, 20:00
alert( formatDate(new Date(new Date - 86400 *
1000)) );
```

Альтернативное решение:

```
function formatDate(date) {
  let dayOfMonth = date.getDate();
  let month = date.getMonth() + 1;
  let year = date.getFullYear();
  let hour = date.getHours();
  let minutes = date.getMinutes();
  let diffMs = new Date() - date;
  let diffSec = Math.round(diffMs / 1000);
  let diffMin = diffSec / 60;
  let diffHour = diffMin / 60;

  // форматирование
  year = year.toString().slice(-2);
  month = month < 10 ? '0' + month : month;
  dayOfMonth = dayOfMonth < 10 ? '0' + dayOfMonth :
dayOfMonth;
  hour = hour < 10 ? '0' + hour : hour;
```

```
  minutes = minutes < 10 ? '0' + minutes : minutes;

  if (diffSec < 1) {
    return 'прямо сейчас';
  } else if (diffMin < 1) {
    return `${diffSec} сек. назад`
  } else if (diffHour < 1) {
    return `${diffMin} мин. назад`
  } else {
    return `${dayOfMonth}.${month}.${year} ${hour}:
${minutes}`
  }
}
```

Открыть решение с тестами в песочнице.

К условию

Формат JSON, метод toJSON

Преобразуйте объект в JSON, а затем обратно в обычный объект

```
let user = {
  name: "Василий Иванович",
  age: 35
};

let user2 =
JSON.parse(JSON.stringify(user));
```

К условию

Исключить обратные ссылки

```
let room = {
  number: 23
};

let meetup = {
  title: "Совещание",
  occupiedBy: [{name: "Иванов"}, {name: "Петров"}],
  place: room
};

room.occupiedBy = meetup;
meetup.self = meetup;

alert( JSON.stringify(meetup, function
replacer(key, value) {
  return (key != "" && value == meetup) ? undefined
: value;
}));

/*
{
  "title":"Совещание",
  "occupiedBy":[{"name":"Иванов"},
{"name":"Петров"}],
  "place":{"number":23}
}
*/
```

Здесь нам также нужно проверить `key ==""`, чтобы исключить первый вызов, где значение `value` равно `meetup`.

К условию

Рекурсия и стек

Вычислить сумму чисел до данного

Решение с помощью цикла:

```
function sumTo(n) {
  let sum = 0;
  for (let i = 1; i <= n; i++) {
    sum += i;
  }
  return sum;
}

alert( sumTo(100) );
```

Решение через рекурсию:

```
function sumTo(n) {
  if (n == 1) return 1;
  return n + sumTo(n - 1);
}

alert( sumTo(100) );
```

Решение по формуле: `sumTo(n) = n*(n+1)/2`:

```
function sumTo(n) {
  return n * (n + 1) / 2;
}

alert( sumTo(100) );
```

P.S. Надо ли говорить, что решение по формуле работает быстрее всех? Это очевидно. Оно использует всего три операции для любого n, а цикл и рекурсия требуют как минимум n операций сложения.

Вариант с циклом – второй по скорости. Он быстрее рекурсии, так как операций сложения столько же, но нет дополнительных вычислительных затрат на организацию

вложенных вызовов. Поэтому рекурсия в данном случае работает медленнее всех.

P.P.S. Некоторые движки поддерживают оптимизацию «хвостового вызова»: если рекурсивный вызов является самым последним в функции (как в `sumTo` выше), то внешней функции не нужно будет возобновлять выполнение и не нужно запоминать контекст его выполнения. В итоге требования к памяти снижаются, и сумма `sumTo(100000)` будет успешно вычислена. Но если JavaScript-движок не поддерживает это (большинство не поддерживают), будет ошибка: максимальный размер стека превышен, так как обычно существует ограничение на максимальный размер стека.

К условию

Вычислить факториал

По определению факториал `n!` можно записать как `n * (n-1)!`.

Другими словами, `factorial(n)` можно получить как `n` умноженное на результат `factorial(n-1)`. И результат для `n-1`, в свою очередь, может быть вычислен рекурсивно и так далее до `1`.

```
function factorial(n) {
  return (n != 1) ? n * factorial(n - 1) : 1;
}
```

```
alert( factorial(5) ); // 120
```

Базисом рекурсии является значение `1`. А можно было бы сделать базисом и `0`, однако это добавило рекурсии дополнительный шаг:

```
function factorial(n) {
  return n ? n * factorial(n - 1) : 1;
}

alert( factorial(5) ); // 120
```

К условию

Числа Фибоначчи

Сначала решим через рекурсию.

Числа Фибоначчи рекурсивны по определению:

```
function fib(n) {
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
// fib(77); // вычисляется очень долго
```

При больших значениях `n` такое решение будет работать очень долго. Например, `fib(77)` может повесить браузер на некоторое время, съев все ресурсы процессора.

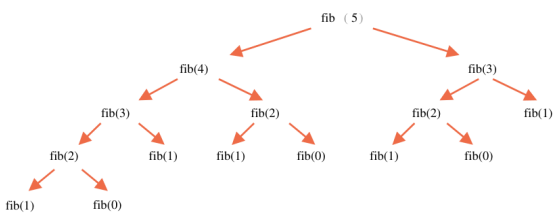
Это потому, что функция порождает обширное дерево вложенных вызовов. При этом ряд значений вычисляется много раз снова и снова.

Например, посмотрим на отрывок вычислений для `fib(5)`:

```
...
fib(5) = fib(4) + fib(3)
fib(4) = fib(3) + fib(2)
...
```

Здесь видно, что значение `fib(3)` нужно одновременно и для `fib(5)` и для `fib(4)`. В коде оно будет вычислено два раза, совершенно независимо.

Полное дерево рекурсии:



Можно заметить, что `fib(3)` вычисляется дважды, а `fib(2)` – трижды. Общее количество вычислений растёт намного быстрее, чем `n`, что делает его огромным даже для `n=77`.

Можно это оптимизировать, запоминая уже вычисленные значения: если значение, скажем, `fib(3)` вычислено однажды, затем мы просто переиспользуем это значение для последующих вычислений.

Другим вариантом было бы отказаться от рекурсии и использовать совершенно другой алгоритм на основе цикла.

Вместо того, чтобы начинать с `n` и вычислять необходимые предыдущие значения, можно написать цикл, который начнёт с `1` и `2`, затем из них получит `fib(3)` как их сумму, затем `fib(4)` как сумму предыдущих значений, затем `fib(5)` и так далее, до финального результата. На каждом шаге нам нужно помнить только значения двух предыдущих чисел последовательности.

Вот детальные шаги нового алгоритма.

Начало:

```
// a = fib(1), b = fib(2), эти значения по
определению равны 1
let a = 1, b = 1;

// получим c = fib(3) как их сумму
let c = a + b;

/* теперь у нас есть fib(1), fib(2), fib(3)
a  b  c
1, 1, 2
*/
```

Теперь мы хотим получить `fib(4) = fib(2) + fib(3)`.

Переставим переменные: `a,b`, присвоим значения `fib(2),fib(3)`, тогда `c` можно получить как их сумму:

```
a = b; // теперь a = fib(2)
b = c; // теперь b = fib(3)
c = a + b; // c = fib(4)

/* имеем последовательность:
   a  b  c
1, 1, 2, 3
```

```
*/
```

Следующий шаг даёт новое число последовательности:

```
a = b; // now a = fib(3)
b = c; // now b = fib(4)
c = a + b; // c = fib(5)

/* последовательность теперь (на одно число
больше):
      a  b  c
1, 1, 2, 3, 5
*/
```

…И так далее, пока не получим искомое значение. Это намного быстрее рекурсии и не требует повторных вычислений.

Полный код:

```
function fib(n) {
  let a = 1;
  let b = 1;
  for (let i = 3; i <= n; i++) {
    let c = a + b;
    a = b;
    b = c;
  }
  return b;
}

alert( fib(3) ); // 2
alert( fib(7) ); // 13
alert( fib(77) ); // 5527939700884757
```

Цикл начинается с `i=3`, потому что первое и второе значения последовательности заданы `a=1, b=1`.

Такой способ называется динамическое программирование снизу вверх.

К условию

Вывод односвязного списка

Решение с использованием цикла

Решение с использованием цикла:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printList(list) {
  let tmp = list;

  while (tmp) {
    alert(tmp.value);
    tmp = tmp.next;
  }

}

printList(list);
```

Обратите внимание, что мы используем временную переменную `tmp` для перемещения по списку. Технически, мы могли бы использовать параметр функции `list` вместо неё:

```
function printList(list) {

  while() {
```

```
    alert(list.value);
    list = list.next;
  }

}
list
```

…Но это было бы неблагоразумно. В будущем нам может понадобиться расширить функцию, сделать что-нибудь ещё со списком. Если мы меняем `list`, то теряем такую возможность.

Говоря о хороших именах для переменных, `list` здесь – это сам список, его первый элемент. Так и должно быть, это просто и понятно.

С другой стороны, `tmp` используется исключительно для обхода списка, как `i` в цикле `for`.

Решение через рекурсию

Рекурсивный вариант `printList(list)` следует простой логике: для вывода списка мы должны вывести текущий `list`, затем сделать то же самое для `list.next`:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
```

```
  }
};

function printList(list) {

  alert(list.value); // выводим текущий элемент

  if (list.next) {
    printList(list.next); // делаем то же самое для
остальной части списка
  }

}

printList(list);
```

Какой способ лучше?

Технически, способ с циклом более эффективный. В обеих реализациях делается то же самое, но для цикла не тратятся ресурсы для вложенных вызовов.

С другой стороны, рекурсивный вариант более короткий и, возможно, более простой для понимания.

К условию

Вывод односвязного списка в обратном порядке

С использованием рекурсии

Рекурсивная логика в этом случае немного сложнее.

Сначала надо вывести оставшуюся часть

списка, а *затем* текущий элемент:

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printReverseList(list) {

  if (list.next) {
    printReverseList(list.next);
  }

  alert(list.value);
}

printReverseList(list);
```

С использованием цикла

Вариант с использованием цикла сложнее, чем в предыдущей задаче.

Нет способа сразу получить последнее значение в списке `list`. Мы также не можем «вернуться назад», к предыдущему элементу списка.

Поэтому мы можем сначала перебрать элементы в прямом порядке и запомнить их в массиве, а затем вывести то, что мы запомнили, в обратном порядке:

```
let list = {
```

```
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

function printReverseList(list) {
  let arr = [];
  let tmp = list;

  while (tmp) {
    arr.push(tmp.value);
    tmp = tmp.next;
  }

  for (let i = arr.length - 1; i >= 0; i--) {
    alert( arr[i] );
  }
}

printReverseList(list);
```

Обратите внимание, что рекурсивное решение на самом деле делает то же самое: проходит список, запоминает элементы в цепочке вложенных вызовов (в контексте выполнения), а затем выводит их.

К условию

Замыкание

Независимы ли счётчики?

Ответ: **0,1.**

Функции `counter` и `counter2` созданы разными вызовами `makeCounter`.

Так что у них независимые внешние лексические окружения, у каждого из которых свой собственный `count`.

К условию

Объект счётчика

Несомненно, он отлично будет работать.

Обе вложенные функции были созданы с одним и тем же внешним лексическим окружением, так что они имеют доступ к одной и той же переменной `count`:

```
function Counter() {
  let count = 0;

  this.up = function() {
    return ++count;
  };

  this.down = function() {
    return --count;
  };
}

let counter = new Counter();

alert( counter.up() ); // 1
```

```
alert( counter.up() ); // 2
alert( counter.down() ); // 1
```

К условию

Функция в if

Результатом будет **ошибка**.

Функция `sayHi` объявлена внутри `if`, так что она живёт только внутри этого блока. Снаружи нет `sayHi`.

К условию

Сумма с помощью замыканий

Чтобы вторые скобки заработали, первые – должны вернуть функцию.

Вот так:

```
function sum(a) {

  return function(b) {
    return a + b; // берёт "a" из внешнего лексического окружения
  };

}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1) ); // 4
```

К условию

Фильтрация с помощью функции

Фильтр inBetween

```
function inBetween(a, b) {
  return function(x) {
    return x >= a && x <= b;
  };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inBetween(3, 6)) ); // 3,4,5,6
```

Фильтр inArray

```
function inArray(arr) {
  return function(x) {
    return arr.includes(x);
  };
}

let arr = [1, 2, 3, 4, 5, 6, 7];
alert( arr.filter(inArray([1, 2, 10])) ); // 1,2
```

Открыть решение с тестами в песочнице.

К условию

Сортировать по полю

```
let users = [
  { name: "John", age: 20, surname: "Johnson" },
  { name: "Pete", age: 18, surname: "Peterson" },
  { name: "Ann", age: 19, surname: "Hathaway" }
];



users.sort(byField('name'));
users.forEach(user => alert(user.name)); // Ann,
John, Pete
```

```
users.sort(byField('age'));
users.forEach(user => alert(user.name)); // Pete,
Ann, John
function byField(field) {
  return (a, b) => a[field] > b[field]
? 1 : -1;
}
```

К условию

Армия функций

Давайте посмотрим, что происходит внутри `makeArmy`, и решение станет очевидным.

1. Она создаёт пустой массив `shooters`:

   ```
   let shooters = [];
   ```

2. В цикле заполняет его `shooters.push(function...)`.

   Каждый элемент – это функция, так что получится такой массив:

   ```
   shooters = [
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); },
     function () { alert(i); }
   ];
   ```

3. Функция возвращает массив.

Позже вызов `army[5]()` получит элемент `army[5]` из массива (это будет функция) и вызовет её.

Теперь, почему все эти функции показывают одно и то же?

Всё потому, что внутри функций `shooter` нет локальной переменной `i`. Когда вызывается такая функция, она берёт `i` из своего внешнего лексического окружения.

Какое будет значение у `i`?

Если мы посмотрим в исходный код:

```
function makeArmy() {
  ...
  let i = 0;
  while (i < 10) {
    let shooter = function() { // функция shooter
      alert( i ); // должна выводить порядковый
номер
    };
    ...
  }
  ...
}
```

…Мы увидим, что оно живёт в лексическом окружении, связанном с текущим вызовом `makeArmy()`. Но, когда вызывается `army[5]()`, `makeArmy` уже завершила свою работу, и последнее значение `i`: 10 (конец цикла `while`).

Как результат, все функции `shooter` получат

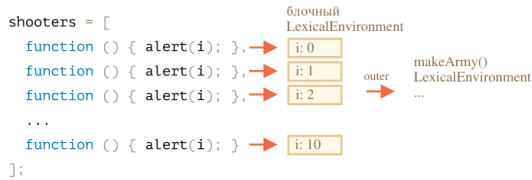
одно и то же из внешнего окружения: последнее значение `i=10`.

Мы можем это исправить, переместив определение переменной в цикл:

```
function makeArmy() {

  let shooters = [];


    let shooter = function() { // функция shooter
      alert( i ); // должна выводить порядковый
номер
    };
    shooters.push(shooter);


  }

  return shooters;
}

let army = makeArmy();

army[0](); // 0
army[5](); // 5
   for(let i = 0; i < 10; i++) {
```

Теперь она работает правильно, потому что каждый раз, когда выполняется блок кода `for (let i=0...) {...}`, для него создаётся новое лексическое окружение с соответствующей переменной `i`.

Так что значение `i` теперь живёт немного ближе. Не в лексическом окружении `makeArmy()`, а в лексическом окружении, которое соответствует текущей итерации цикла. Вот почему теперь она работает.



Здесь мы переписали `while` в `for`.

Можно использовать другой трюк, давайте рассмотрим его для лучшего понимания предмета:

```
function makeArmy() {
  let shooters = [];

  let i = 0;
  while (i < 10) {

    let shooter = function() { // функция shooter
      alert(  ); // должна выводить порядковый
номер
    };
    shooters.push(shooter);
    i++;
  }

  return shooters;
}

let army = makeArmy();

army[0](); // 0
army[5](); // 5
     let j = i;
j
```

Цикл `while` так же, как и `for`, создаёт новое лексическое окружение для каждой итерации. Так что тут мы хотим убедиться, что он получит правильное значение для `shooter`.

Мы копируем `let j = i`. Это создаёт

локальную для итерации переменную `j` и копирует в неё `i`. Примитивы копируются «по значению», поэтому мы получаем совершенно независимую копию `i`, принадлежащую текущей итерации цикла.

Открыть решение с тестами в песочнице.

К условию

Объект функции, NFE

Установка и уменьшение значения счётчика

В решении использована локальная переменная `count`, а методы сложения записаны прямо в `counter`. Они разделяют одно и то же лексическое окружение и также имеют доступ к текущей переменной `count`.

```
function makeCounter() {
  let count = 0;

  function counter() {
    return count++;
  }

  counter.set = value => count = value;

  counter.decrease = () => count--;
```

```
  return counter;
}
```

Открыть решение с тестами в песочнице.

К условию

Сумма с произвольным количеством скобок

1. В общем, чтобы это *хоть как-нибудь* заработало, результат, возвращаемый `sum`, должен быть функцией.
2. Между вызовами эта функция должна удерживать в памяти текущее значение счётчика.
3. Согласно заданию, функция должна преобразовываться в число, когда она используется с оператором `==`. Функции – объекты, так что преобразование происходит, как описано в главе Преобразование объектов в примитивы, поэтому можно создать наш собственный метод, возвращающий число.

Код:

```
function sum(a) {

  let currentSum = a;

  function f(b) {
    currentSum += b;
    return f;
  }
```

```
  f.toString = function() {
    return currentSum;
  };

  return f;
}

alert( sum(1)(2) ); // 3
alert( sum(5)(-1)(2) ); // 6
alert( sum(6)(-1)(-2)(-3) ); // 0
alert( sum(0)(1)(2)(3)(4)(5) ); // 15
```

Пожалуйста, обратите внимание на то, что функция `sum` выполняется лишь однажды и просто возвращает функцию `f`.

Далее, при каждом последующем вызове, `f` суммирует свой аргумент со значением `currentSum` и возвращает себя же.

В последней строке `f` нет никакой рекурсии.

Вот как выглядит рекурсия:

```
function f(b) {
  currentSum += b;
  return f(); // <-- рекурсивный вызов
}
```

В нашем случае мы просто возвращаем функцию, не вызывая её:

```
function f(b) {
  currentSum += b;
  return f; // <-- не вызывает себя. Просто
возвращает
}
```

Функция `f` будет использоваться в последующем вызове и снова возвращать себя

столько раз, сколько будет необходимо. Затем, при использовании в качестве числа или строки, метод `toString` возвращает `currentSum` – число. Также здесь мы можем использовать `Symbol.toPrimitive` или `valueOf` для преобразования.

К условию

Планирование: setTimeout и setInterval

Вывод каждую секунду

Используем `setInterval`:

```
function printNumbers(from, to) {
  let current = from;

  let timerId = setInterval(function() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }, 1000);
}

// использование:
printNumbers(5, 10);
```

Используем рекурсивный `setTimeout`:

```
function printNumbers(from, to) {
  let current = from;
```

```
  setTimeout(function go() {
    alert(current);
    if (current < to) {
      setTimeout(go, 1000);
    }
    current++;
  }, 1000);
}

// использование:
printNumbers(5, 10);
```

Заметим, что в обоих решениях есть начальная задержка перед первым выводом. Она составляет одну секунду (1000мс). Если мы хотим, чтобы функция запускалась сразу же, то надо добавить такой запуск вручную на отдельной строке, вот так:

```
function printNumbers(from, to) {
  let current = from;

  function go() {
    alert(current);
    if (current == to) {
      clearInterval(timerId);
    }
    current++;
  }


  let timerId = setInterval(go, 1000);
}

printNumbers(5, 10);
  go();
```

К условию

Что покажет setTimeout?

Любой вызов `setTimeout` будет выполнен

только после того, как текущий код завершится.

Последним значением `i` будет: `100000000`.

```
let i = 0;

setTimeout(() => alert(i), 100); // 100000000

// предположим, что время выполнения этой функции
>100 мс
for(let j = 0; j < 100000000; j++) {
  i++;
}
```

К условию

Декораторы и переадресация вызова, call/apply

Декоратор-шпион

Здесь мы можем использовать `call.push(args)` для хранения всех аргументов в списке и `f.apply(this, args)` для переадресации вызова.

```
function spy(func) {

  function wrapper(...args) {
    wrapper.calls.push(args);
    return func.apply(this, arguments);
  }

  wrapper.calls = [];

  return wrapper;
```

```
}
```

Открыть решение с тестами в песочнице.

К условию

Задерживающий декоратор

Решение:

```
function delay(f, ms) {

  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };

}

let f1000 = delay(alert, 1000);

f1000("test"); // показывает "test" после 1000 мс
```

Обратите внимание, как здесь используется функция-стрелка. Как мы знаем, функция-стрелка не имеет собственных `this` и `arguments`, поэтому `f.apply(this, arguments)` берет `this` и `arguments` из обёртки.

Если мы передадим обычную функцию, `setTimeout` вызовет её без аргументов и с `this=window` (при условии, что код выполняется в браузере).

Мы всё ещё можем передать правильный `this`, используя промежуточную переменную, но это немного громоздко:

```
function delay(f, ms) {

  return function(...args) {
    let savedThis = this; // сохраняем this в
промежуточную переменную
    setTimeout(function() {
      f.apply(savedThis, args); // используем её
    }, ms);
  };

}
```

Открыть решение с тестами в песочнице.

К условию

Декоратор debounce

```
function debounce(f, ms) {

  let isCooldown = false;

  return function() {
    if (isCooldown) return;

    f.apply(this, arguments);

    isCooldown = true;

    setTimeout(() => isCooldown = false, ms);
  };

}
```

Вызов `debounce` возвращает обёртку.
Возможны два состояния:

- `isCooldown = false` – готова к выполнению.
- `isCooldown = true` – ожидание окончания тайм-аута.

В первом вызове `isCoolDown = false`, поэтому вызов продолжается, и состояние изменяется на `true`.

Пока `isCoolDown` имеет значение `true`, все остальные вызовы игнорируются.

Затем `setTimeout` устанавливает его в `false` после заданной задержки.

Открыть решение с тестами в песочнице.

К условию

Тормозящий (throttling) декоратор

```
function throttle(func, ms) {

  let isThrottled = false,
    savedArgs,
    savedThis;

  function wrapper() {

    if (isThrottled) { // (2)
      savedArgs = arguments;
      savedThis = this;
      return;
    }

    func.apply(this, arguments); // (1)

    isThrottled = true;

    setTimeout(function() {
      isThrottled = false; // (3)
      if (savedArgs) {
        wrapper.apply(savedThis, savedArgs);
        savedArgs = savedThis = null;
      }
    }, ms);
  }
```

```
  return wrapper;
}
```

Вызов `throttle(func, ms)` возвращает `wrapper`.

1. Во время первого вызова обёртка просто вызывает `func` и устанавливает состояние задержки (`isThrottled = true`).
2. В этом состоянии все вызовы запоминаются в `saveArgs / saveThis`. Обратите внимание, что контекст и аргументы одинаково важны и должны быть запомнены. Они нам нужны для того, чтобы воспроизвести вызов позднее.
3. … Затем по прошествии `ms` миллисекунд срабатывает `setTimeout`. Состояние задержки сбрасывается (`isThrottled = false`). И если мы проигнорировали вызовы, то «обёртка» выполняется с последними запомненными аргументами и контекстом.

На третьем шаге выполняется не `func`, а `wrapper`, потому что нам нужно не только выполнить `func`, но и ещё раз установить состояние задержки и таймаут для его сброса.

Открыть решение с тестами в песочнице.

К условию

Связанная функция как метод

Ответ: `null`.

```
function f() {
  alert( this ); // null
}

let user = {
  g: f.bind(null)
};

user.g();
```

Контекст связанной функции жёстко фиксирован. Изменить однажды привязанный контекст уже нельзя.

Так что хоть мы и вызываем `user.g()`, внутри исходная функция будет вызвана с `this=null`. Однако, функции `g` совершенно без разницы, какой `this` она получила. Её единственное предназначение – это передать вызов в `f` вместе с аргументами и ранее указанным контекстом `null`, что она и делает.

Таким образом, когда мы запускаем `user.g()`, исходная функция вызывается с `this=null`.

К условию

Повторный bind

Ответ: **Вася**.

```
function f() {
  alert(this.name);
}

f = f.bind( {name: "Вася"} ).bind( {name: "Петя"}
);

f(); // Вася
```

Экзотический объект bound function, возвращаемый при первом вызове `f.bind(...)`, запоминает контекст (и аргументы, если они были переданы) только во время создания.

Следующий вызов `bind` будет устанавливать контекст уже для этого объекта. Это ни на что не повлияет.

Можно сделать новую привязку, но нельзя изменить существующую.

К условию

Свойство функции после bind

Ответ: `undefined`.

Результатом работы `bind` является другой объект. У него уже нет свойства `test`.

К условию

Исправьте функцию, теряющую "this"

Ошибка происходит потому, что `askPassword` получает функции `loginOk/loginFail` без контекста.

Когда они вызываются, то, естественно, `this=undefined`.

Используем `bind`, чтобы передать в `askPassword` функции `loginOk/loginFail` с уже привязанным контекстом:

```
function askPassword(ok, fail) {
  let password = prompt("Password?", '');
  if (password == "rockstar") ok();
  else fail();
}

let user = {
  name: 'Вася',

  loginOk() {
    alert(`${this.name} logged in`);
  },

  loginFail() {
    alert(`${this.name} failed to log in`);
  },

};

askPassword(user.loginOk.bind(user),
user.loginFail.bind(user));
```

Теперь всё работает корректно.

Альтернативное решение – сделать функции-обёртки над `user.loginOk/loginFail`:

```
//...
askPassword(() => user.loginOk(), () =>
user.loginFail());
```

Обычно это также работает и хорошо выглядит. Но может не сработать в более сложных ситуациях, а именно – когда значение переменной `user` меняется между вызовом `askPassword` и выполнением `() => user.loginOk()`.

К условию

Использование частично применённой функции для логина

1. Можно использовать стрелочную функцию-обёртку:

   ```
   askPassword(() => user.login(true), () =>
   user.login(false));
   ```

   Теперь она получает `user` извне и нормально выполняется.

2. Или же можно создать частично применённую функцию на основе `user.login`, которая использует объект `user` в качестве контекста и получает соответствующий первый аргумент:

```
askPassword(user.login.bind(user, true),
user.login.bind(user, false));
```

К условию

Прототипное наследование

Работа с прототипами

1. `true`, берётся из `rabbit`.
2. `null`, берётся из `animal`.
3. `undefined`, такого свойства больше нет.

К условию

Алгоритм поиска

1. Добавим свойство `__proto__`:

```
let head = {
  glasses: 1
};

let table = {
  pen: 3,
  __proto__: head
};

let bed = {
  sheet: 1,
  pillow: 2,
  __proto__: table
```

```
};

let pockets = {
  money: 2000,
  __proto__: bed
};

alert( pockets.pen ); // 3
alert( bed.glasses ); // 1
alert( table.money ); // undefined
```

2. С точки зрения производительности, для современных движков неважно, откуда берётся свойство – из объекта или из прототипа. Они запоминают, где было найдено свойство, и повторно используют его в следующем запросе.

   Например, при обращении к `pockets.glasses` они запомнят, что нашли `glasses` в `head`, и в следующий раз будут искать там же. Они достаточно умны, чтобы при изменениях обновлять внутренний кеш, поэтому такая оптимизация безопасна.

К условию

Куда будет произведена запись?

Ответ: `rabbit`.

Поскольку `this` – это объект, который стоит перед точкой, `rabbit.eat()` изменяет объект `rabbit`.

Поиск свойства и исполнение кода – два разных

процесса. Сначала осуществляется поиск метода `rabbit.eat` в прототипе, а затем этот метод выполняется с `this=rabbit`.

К условию

Почему наедаются оба хомяка?

Давайте внимательно посмотрим, что происходит при вызове `speedy.eat("apple")`.

1. Сначала в прототипе (`=hamster`) находится метод `speedy.eat`, а затем он выполняется с `this=speedy` (объект перед точкой).

2. Затем в `this.stomach.push()` нужно найти свойство `stomach` и вызвать для него `push`. Движок ищет `stomach` в `this` (`=speedy`), но ничего не находит.

3. Он идёт по цепочке прототипов и находит `stomach` в `hamster`.

4. И вызывает для него `push`, добавляя еду в *живот прототипа*.

Получается, что у хомяков один живот на двоих!

И при `lazy.stomach.push(...)` и при `speedy.stomach.push()`, свойство `stomach` берётся из прототипа (так как его нет в самом

объекте), затем в него добавляются данные.

Обратите внимание, что этого не происходит при простом присваивании `this.stomach=`:

```
let hamster = {
  stomach: [],

  eat(food) {

  }
};

let speedy = {
   __proto__: hamster
};

let lazy = {
  __proto__: hamster
};

// Шустрый хомяк нашёл еду
speedy.eat("apple");
alert( speedy.stomach ); // apple

// Живот ленивого хомяка пуст
alert( lazy.stomach ); // <ничего>
     // присвоение значения
this.stomach вместо вызова
this.stomach.push
     this.stomach = [food];
```

Теперь всё работает правильно, потому что `this.stomach=` не ищет свойство `stomach`. Значение записывается непосредственно в объект `this`.

Также мы можем полностью избежать проблемы, если у каждого хомяка будет собственный живот:

```
let hamster = {
  stomach: [],
```

```
  eat(food) {
    this.stomach.push(food);
  }
};

let speedy = {
  __proto__: hamster,

};

let lazy = {
  __proto__: hamster,

};

// Шустрый хомяк нашёл еду
speedy.eat("apple");
alert( speedy.stomach ); // apple

// Живот ленивого хомяка пуст
alert( lazy.stomach ); // <ничего>
```

```
  stomach: []
  stomach: []
```

Все свойства, описывающие состояние объекта (как свойство `stomach` в примере выше), рекомендуется записывать в сам этот объект. Это позволяет избежать подобных проблем.

К условию

F.prototype

Изменяем "prototype"

Ответы:

1. `true`.

   Присвоение нового значения свойству `Rabbit.prototype` влияет на `[[Prototype]]` вновь создаваемых объектов, но не на прототип уже существующих.

2. `false`.

   Объекты присваиваются по ссылке. Не создаётся копия `Rabbit.prototype`, это всегда один объект, на который ссылается и `Rabbit.prototype`, и `[[Prototype]]` объекта `rabbit`.

   Таким образом, когда мы изменяем этот объект по одной ссылке, изменения видны и по другой.

3. `true`.

   Операция `delete` применяется к свойствам конкретного объекта, на котором она вызвана. Здесь `delete rabbit.eats` пытается удалить свойство `eats` из объекта `rabbit`, но его там нет. Таким образом, просто ничего не произойдёт.

4. `undefined`.

Свойство `eats` удалено из прототипа, оно больше не существует.

К условию

Создайте новый объект с помощью уже существующего

Мы можем использовать такой способ, если мы уверены в том, что свойство `"constructor"` существующего объекта имеет корректное значение.

Например, если мы не меняли `"prototype"`, используемый по умолчанию, то код ниже, без сомнений, сработает:

```
function User(name) {
  this.name = name;
}

let user = new User('John');
let user2 = new user.constructor('Pete');

alert( user2.name ); // Pete (сработало!)
```

Всё получилось, потому что `User.prototype.constructor == User`.

…Но если кто-то перезапишет `User.prototype` и забудет заново назначить свойство `"constructor"`, чтобы оно указывало на `User`, то ничего не выйдет.

Например:

```
function User(name) {
  this.name = name;
}


let user = new User('John');
let user2 = new user.constructor('Pete');

alert( user2.name ); // undefined
User.prototype = {}; // (*)
```

Почему `user2.name` приняло значение `undefined`?

Рассмотрим, как отработал вызов `new user.constructor('Pete')`:

1. Сначала ищется свойство `constructor` в объекте `user`. Не нашлось.
2. Потом задействуется поиск по цепочке прототипов. Прототип объекта `user` – это `User.prototype`, и там тоже нет искомого свойства.
3. Значение `User.prototype` – это пустой объект `{}`, чей прототип – `Object.prototype`. `Object.prototype.constructor == Object`. Таким образом, свойство `constructor` всё-таки найдено.

Наконец срабатывает `let user2 = new Object('Pete')`, но конструктор `Object` игнорирует аргументы, он всегда создаёт пустой объект: `let user2 = {}` – это как раз то, чему равен `user2` в итоге.

К условию

Встроенные прототипы

Добавить функциям метод "f.defer(ms)"

```
Function.prototype.defer = function(ms) {
  setTimeout(this, ms);
};

function f() {
  alert("Hello!");
}

f.defer(1000); // выведет "Hello!" через 1 секунду
```

К условию

Добавьте функциям декорирующий метод "defer()"

```
Function.prototype.defer = function(ms) {
  let f = this;
  return function(...args) {
    setTimeout(() => f.apply(this, args), ms);
  }
};

// check it
function f(a, b) {
  alert( a + b );
}

f.defer(1000)(1, 2); // выведет 3 через 1 секунду.
```

К условию

Методы прототипов, объекты без свойства __proto__

Добавьте toString в словарь

В методе можно получить все перечисляемые ключи с помощью `Object.keys` и вывести их список.

Чтобы сделать toString неперечисляемым, давайте определим его, используя дескриптор свойства. Синтаксис `Object.create` позволяет нам добавить в объект дескрипторы свойств как второй аргумент.

```
dictionary.apple = "Apple";
dictionary.__proto__ = "test";

// apple и __proto__ выведены в цикле
for(let key in dictionary) {
  alert(key); // "apple", затем "__proto__"
}

// список свойств, разделённых запятой, выведен с
помощью toString
alert(dictionary); // "apple,__proto__"
let dictionary = Object.create(null, {
  toString: { // определяем свойство
toString
    value() { // значение -- это
```

```
функция
      return Object.keys(this).join();
    }
  }
});
```

Когда мы создаём свойство с помощью дескриптора, все флаги по умолчанию имеют значение `false`. Таким образом, в коде выше `dictionary.toString` – неперечисляемое свойство.

Смотрите главу Флаги и дескрипторы свойств для ознакомления.

К условию

Разница между вызовами

В первом вызове `this == rabbit`, во всех остальных `this` равен `Rabbit.prototype`, так как это объект перед точкой.

Так что только первый вызов выведет `Rabbit`, а остальные – `undefined`:

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert( this.name );
}

let rabbit = new Rabbit("Rabbit");

rabbit.sayHi();                         // Rabbit
Rabbit.prototype.sayHi();               // undefined
```

```
Object.getPrototypeOf(rabbit).sayHi(); // undefined
rabbit.__proto__.sayHi();              // undefined
```

К условию

Класс: базовый синтаксис

Перепишите класс

```
class Clock {
  constructor({ template }) {
    this.template = template;
  }

  render() {
    let date = new Date();

    let hours = date.getHours();
    if (hours < 10) hours = '0' + hours;

    let mins = date.getMinutes();
    if (mins < 10) mins = '0' + mins;

    let secs = date.getSeconds();
    if (secs < 10) secs = '0' + secs;

    let output = this.template
      .replace('h', hours)
      .replace('m', mins)
      .replace('s', secs);

    console.log(output);
  }

  stop() {
    clearInterval(this.timer);
  }

  start() {
    this.render();
```

```
    this.timer = setInterval(() => this.render(),
1000);
  }
}


let clock = new Clock({template: 'h:m:s'});
clock.start();
```

Открыть решение в песочнице.

К условию

Наследование классов

Ошибка создания экземпляра класса

Ошибка возникает потому, что конструктор дочернего класса должен вызывать `super()`.

Вот правильный код:

```
class Animal {

  constructor(name) {
    this.name = name;
  }

}

class Rabbit extends Animal {
  constructor(name) {

    this.created = Date.now();
```

```
  }
}

alert(rabbit.name); // White Rabbit
    super(name);
let rabbit = new Rabbit("Белый
кролик"); // ошибки нет
```

К условию

Улучшенные часы

```
class ExtendedClock extends Clock {
  constructor(options) {
    super(options);
    let { precision=1000 } = options;
    this.precision = precision;
  }

  start() {
    this.render();
    this.timer = setInterval(() => this.render(),
this.precision);
  }
};
```

Открыть решение в песочнице.

К условию

Класс расширяет объект?

Сперва давайте разберёмся, почему код не работает.

Причина становится очевидна, если мы попытаемся запустить его. Унаследованный конструктор класса должен вызывать `super()`. В противном случае `"this"` будет не определён.

Решение:

```
class Rabbit extends Object {
  constructor(name) {

    this.name = name;
  }
}

let rabbit = new Rabbit("Кроль");

alert( rabbit.hasOwnProperty('name') ); // true
    super(); // надо вызвать конструктор родителя, когда наследуемся
```

Но это ещё не все.

Даже после исправления есть важное различие между `"class Rabbit extends Object"` и `class Rabbit`.

Как мы знаем, синтаксис «extends» устанавливает 2 прототипа:

1. Между `"prototype"` функций-конструкторов (для методов)
2. Между самими функциями-конструкторами (для статических методов).

В нашем случае, для `class Rabbit extends Object` это значит:

```
class Rabbit extends Object {}

alert( Rabbit.prototype.__proto__ ===
Object.prototype ); // (1) true
alert( Rabbit.__proto__ === Object ); // (2) true
```

Таким образом, `Rabbit` предоставляет доступ к статическим методам `Object` через `Rabbit`, например:

```
class Rabbit extends Object {}

// обычно мы вызываем
Object.getOwnPropertyNames
alert ( Rabbit.getOwnPropertyNames({a:
1, b: 2})); // a,b
```

Но если явно не наследуем от объекта, то для `Rabbit.__ proto__` не установлено значение `Object`.

Пример:

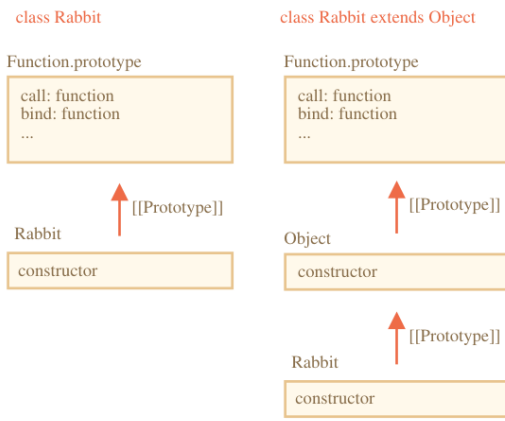
```
class Rabbit {}

alert( Rabbit.prototype.__proto__ ===
Object.prototype ); // (1) true
```

```
alert( Rabbit.__proto__ === Object ); // (2) false
(!)
alert( Rabbit.__proto__ === Function.prototype );
// как у каждой функции по умолчанию

// ошибка – нет такой функции у Rabbit
alert ( Rabbit.getOwnPropertyNames({a:
1, b: 2})); // Error
```

Таким образом, в этом случае у `Rabbit` нет доступа к статическим методам `Object`.

Кстати, у `Function.prototype` есть «общие» методы, такие как `call, bind` и т. д. Они в конечном итоге доступны в обоих случаях, потому что для встроенного конструктора `Object Object.__proto__ === Function.prototype`.

Пример на картинке:



Короче говоря, есть два отличия:

class Rabbit	class Rabbit extends Object

–	необходимо вызвать `super()` в конструкторе
`Rabbit.__proto__ === Function.prototype`	`Rabbit.__proto__ === Object`

К условию

Проверка класса: "instanceof"

Странный instanceof

Да, действительно, выглядит странно.

Но `instanceof` не учитывает саму функцию при проверке, а только `prototype`, который проверяется на совпадения в прототипной цепочке.

И в данном примере `a.__proto__ == B.prototype`, так что `instanceof` возвращает `true`.

Таким образом, по логике `instanceof`, именно `prototype` в действительности определяет тип, а не функция-конструктор.

К условию

Обработка ошибок, "try..catch"

Finally или просто код?

Разница в поведении станет очевидной, если рассмотреть код внутри функции.

Поведение будет различным, если управление каким-то образом выпрыгнет из `try..catch`.

Например, когда есть `return` внутри `try..catch`. Секция `finally` работает в любом случае при *любом* выходе из `try..catch`, даже через `return`: сразу после того как `try..catch` выполнится, но до того, как вызывающий код получит контроль.

```
function f() {
  try {
    alert('начало');

  } catch (e) {
    /// ...
  } finally {
    alert('очистка!');
  }
}

f(); // очистка!
     return "result";
```

…Или когда есть `throw`, например, как здесь:

```
function f() {
  try {
    alert('начало');
    throw new Error("ошибка");
  } catch (e) {
    // ...
    if("не могу обработать ошибку") {

    }

  } finally {
    alert('очистка!')
  }
}

f(); // очистка!
        throw e;
```

Именно `finally` гарантирует очистку. Если мы просто поместим код в конце `f`, то он не выполнится в описанных ситуациях.

К условию

Пользовательские ошибки, расширение Error

Наследование от SyntaxError

```
class FormatError extends SyntaxError {
  constructor(message) {
    super(message);
    this.name = "FormatError";
  }
}
```

```
let err = new FormatError("ошибка форматирования");

alert( err.message ); // ошибка форматирования
alert( err.name ); // FormatError
alert( err.stack ); // stack

alert( err instanceof SyntaxError ); // true
```

К условию

Введение: колбэки

Анимация круга с помощью колбэка

Открыть решение в песочнице.

К условию

Промисы

Можно ли "перевыполнить" промис?

Вывод будет: `1`.

Второй вызов `resolve` будет проигнорирован, поскольку учитывается только первый вызов `reject/resolve`. Все последующие вызовы – игнорируются.

К условию

Задержка на промисах

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => alert('выполнилось через 3 секунды'));
```

Заметьте, что `resolve` вызывается без аргументов. Мы не возвращаем из `delay` ничего, просто гарантируем задержку.

К условию

Анимация круга с помощью промиса

Открыть решение в песочнице.

К условию

Промисы: сравните then и catch

Короткий ответ: **нет, они не эквивалентны**:

Разница в том, что если ошибка произойдёт в `f1`, то она будет обработана в `.catch` в этом примере:

```
promise
  .then(f1)
  .catch(f2);
```

…но не в этом:

```
promise
  .then(f1, f2);
```

Ошибка передаётся по цепочке, но во втором примере нет продолжения цепочки после `f1`.

Другими словами, `.then` передаёт результат или ошибку следующему блоку `.then/catch`. Так как в первом примере в цепочке далее имеется блок `catch`, а во втором – нет, то ошибка в нём останется необработанной.

К условию

Ошибка в setTimeout

Ответ: **нет, не выполнится**:

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

Как было сказано в главе, здесь присутствует "скрытый `try..catch`" вокруг кода функции. Поэтому обрабатываются все синхронные ошибки.

В данном примере ошибка генерируется не по ходу выполнения кода, а позже. Поэтому промис не может обработать её.

К условию

Перепишите, используя async/await

Комментарии к решению под кодом:

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)

  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }

  throw new Error(response.status);
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

Комментарии:

1. Функция `loadJson` теперь асинхронная.
2. Все `.then` внутри неё заменены на `await`.
3. Можно было бы просто вернуть промис во внешний код `return response.json()`, вот так:

   ```
   if (response.status == 200) {
     return response.json(); // (3)
   }
   ```

   Тогда внешнему коду пришлось бы получать результат промиса самостоятельно (через `.then` или `await`). В нашем варианте это не обязательно.
4. Выброшенная из `loadJson` ошибка перехватывается с помощью `.catch`. Здесь нельзя использовать `await loadJson(…)`, так как мы находимся не в теле функции

async.

К условию

Перепишите, используя async/await

В этой задаче нет ничего сложного. Нужно заменить `.catch` на `try...catch` внутри `demoGithubUser` и добавить `async/await`, где необходимо:

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}
`);
    this.name = 'HttpError';
    this.response = response;
  }
}

async function loadJson(url) {
  let response = await fetch(url);
  if (response.status == 200) {
    return response.json();
  } else {
    throw new HttpError(response);
  }
}

// Запрашивать логин, пока github не вернёт
существующего пользователя.
async function demoGithubUser() {

  let user;
  while(true) {
    let name = prompt("Введите логин?", "iliakan");

    try {
      user = await loadJson(`https://
api.github.com/users/${name}`);
      break; // ошибок не было, выходим из цикла
    } catch(err) {
      if (err instanceof HttpError &&
```

```
err.response.status == 404) {
        // после alert начнётся новая итерация
цикла
        alert("Такого пользователя не существует,
пожалуйста, повторите ввод.");
      } else {
        // неизвестная ошибка, пробрасываем её
        throw err;
      }
    }
  }


  alert(`Полное имя: ${user.name}.`);
  return user;
}

demoGithubUser();
```

К условию

Вызовите async–функцию из "обычной"

Это тот случай, когда понимание внутреннего устройства работы `async/await` очень кстати.

Здесь нужно думать о вызове функции `async`, как о промисе. И просто воспользоваться `.then`:

```
async function wait() {
  await new Promise(resolve => setTimeout(resolve,
1000));

  return 10;
}

function f() {
  // покажет 10 через 1 секунду

}
```

```
f();
  wait().then(result =>
alert(result));
```

К условию

Генераторы

Псевдослучайный генератор

```
function* pseudoRandom(seed) {
  let value = seed;

  while(true) {
    value = value * 16807 % 2147483647
    yield value;
  }

};

let generator = pseudoRandom(1);

alert(generator.next().value); // 16807
alert(generator.next().value); // 282475249
alert(generator.next().value); // 1622650073
```

Пожалуйста, обратите внимание, то же самое можно сделать с помощью обычной функции, такой как эта:

```
function pseudoRandom(seed) {
  let value = seed;

  return function() {
    value = value * 16807 % 2147483647;
    return value;
```

```
  }
}

let generator = pseudoRandom(1);

alert(generator()); // 16807
alert(generator()); // 282475249
alert(generator()); // 1622650073
```

Это также работает. Но тогда мы потеряем возможность перебора с помощью `for..of` и использования композиции генераторов, которая тоже может быть полезна.

Открыть решение с тестами в песочнице.

К условию

Proxy и Reflect

Ошибка при чтении несуществующего свойства

```
let user = {
  name: "John"
};

function wrap(target) {
  return new Proxy(target, {
    get(target, prop, receiver) {
      if (prop in target) {
        return Reflect.get(target, prop, receiver);
      } else {
        throw new ReferenceError(`Свойство не
существует: "${prop}"`)
```

```
        }
      }
    });
}

user = wrap(user);

alert(user.name); // John
alert(user.age); // Ошибка: Свойство не существует
```

К условию

Получение элемента массива с отрицательной позиции

```
let array = [1, 2, 3];

array = new Proxy(array, {
  get(target, prop, receiver) {
    if (prop < 0) {
      // даже если обращение к элементу идёт как
arr[1]
      // prop является строкой, нужно преобразовать
её к числу
      prop = +prop + target.length;
    }
    return Reflect.get(target, prop, receiver);
  }
});


alert(array[-1]); // 3
alert(array[-2]); // 2
```

К условию

Observable

Решение состоит из двух частей:

1. При вызове `.observe(handler)` нам

нужно где-то сохранить обработчик, чтобы вызвать его позже. Можно хранить обработчики прямо в объекте, создав в нём для этого свой символьный ключ.

2. Нам нужен прокси с ловушкой `set`, чтобы вызывать обработчики при изменении свойств.

```
let handlers = Symbol('handlers');

function makeObservable(target) {
  // 1. Создадим хранилище обработчиков
  target[handlers] = [];

  // положим туда функции-обработчики для вызовов в
будущем
  target.observe = function(handler) {
    this[handlers].push(handler);
  };

  // 2. Создадим прокси для реакции на изменения
  return new Proxy(target, {
    set(target, property, value, receiver) {
      let success = Reflect.set(...arguments); //
перенаправим операцию к оригинальному объекту
      if (success) { // если не произошло ошибки
при записи свойства
        // вызовем обработчики
        target[handlers].forEach(handler =>
handler(property, value));
      }
      return success;
    }
  });
}

let user = {};

user = makeObservable(user);

user.observe((key, value) => {
  alert(`SET ${key}=${value}`);
});

user.name = "John";
```

К условию

Eval: выполнение строки кода

Eval-калькулятор

Давайте будем использовать `eval` для вычисления арифметических выражений:

```
let expr = prompt("Введите арифметическое выражение:", '2*3+2');

alert( eval(expr) );
```

Пользователь может ввести любой текст или код.

В целях безопасности ограничимся только арифметическими операциями, проверяя переданное `expr` с помощью регулярного выражения, чтобы удостовериться, что в нём содержатся только цифры и соответствующие операторы.

К условию

Побитовый оператор и значение

1. Операция `a^b` ставит бит результата в `1`, если на соответствующей битовой позиции в `a` или `b` (но не одновременно) стоит `1`.

   Так как в `0` везде стоят нули, то биты берутся в точности как во втором аргументе.

2. Первое побитовое НЕ `~` превращает `0` в `1`, а `1` в `0`. А второе НЕ превращает ещё раз, в итоге получается как было.

К условию

Проверка, целое ли число

Один из вариантов такой функции:

```
function isInteger(num) {
  return (num ^ 0) === num;
}

alert( isInteger(1) ); // true
alert( isInteger(1.5) ); // false
alert( isInteger(-0.5) ); // false
```

Обратите внимание: `num^0` – в скобках! Это потому, что приоритет операции `^` очень низкий. Если не поставить скобку, то `===` сработает раньше. Получится `num ^ (0 === num)`, а это

уже совсем другое дело.

К условию

Симметричны ли операции ^, |, &?

Операция над числами, в конечном итоге, сводится к битам.

Посмотрим, можно ли поменять местами биты слева и справа.

Например, таблица истинности для `^`:

a	b	результат
0	0	0
0	1	1
1	0	1
1	1	0

Случаи `0^0` и `1^1` заведомо не изменятся при перемене мест, поэтому нас не интересуют. А вот `0^1` и `1^0` эквивалентны и равны `1`.

Аналогично можно увидеть, что и другие операторы симметричны.

Ответ: **да**.

К условию

Почему результат разный?

Всё дело в том, что побитовые операции преобразуют число в 32-битное целое.

Обычно число в JavaScript имеет 64-битный формат с плавающей точкой. При этом часть битов (52) отведены под цифры, часть (11) отведены под хранение номера позиции, на которой стоит десятичная точка, и один бит – знак числа.

Это означает, что максимальное целое число, которое можно хранить, занимает 52 бита.

Число 12345678912345 в двоичном виде: 101100111010011100111100111001011011010011001 (44 цифры).

Побитовый оператор ^ преобразует его в 32-битное путём отбрасывания десятичной точки и «лишних» старших цифр. При этом, так как число большое и старшие биты здесь ненулевые, то, естественно, оно изменится.

Вот ещё пример:

```
// в двоичном виде 10000000000000000000000000000000
(31 цифры)
alert( Math.pow(2, 30) ); // 1073741824
alert( Math.pow(2, 30) ^ 0 ); // 1073741824, всё
ок, длины хватает

// в двоичном виде
100000000000000000000000000000000 (33 цифры)
alert( Math.pow(2, 32) ); // 4294967296
```

```
alert( Math.pow(2, 32) ^ 0 ); // 0, отброшены
старшие цифры, остались нули

// пограничный случай
// в двоичном виде 10000000000000000000000000000000
(32 цифры)
alert( Math.pow(2, 31) ); // 2147483648
alert( Math.pow(2, 31) ^ 0 ); // -2147483648,
ничего не отброшено,
// но первый бит 1 теперь стоит в начале числа и
является знаковым
```

К условию

Intl: интернационализация в JavaScript

Отсортируйте массив с буквой ё

Здесь подойдут стандартные параметры сравнения:

```
let animals = ["тигр", "ёж", "енот", "ехидна",
"АИСТ", "ЯК"];


alert( animals ); // АИСТ,ёж,енот,ехидна,тигр,ЯК
let collator = new Intl.Collator();
animals.sort(function(a, b) {
  return collator.compare(a, b);
});
```

А вот, что было бы при обычном вызове `sort()`:

```
let animals = ["тигр", "ёж", "енот", "ехидна",
"АИСТ", "ЯК"];

alert( animals.sort() ); //
АИСТ,ЯК,енот,ехидна,тигр,ёж
```

К условию